1989

Parallel architectures for solving combinatorial problems of logic design

Phuong Minh Ho
Portland State University

AN ABSTRACT OF THE THESIS OF Phuong Minh Ho for the Master of Science in Electrical and Computer Engineering presented August 7, 1989.

Title: Parallel Architectures for Solving Combinatorial Problems of Logic Design.

APPROVED BY MEMBERS OF THE THESIS COMMITTEE:

[redacted]

Marek Perkowski, Chairman

[redacted]

Michael Driscoll

[redacted]

Robert Daasch

[redacted]

Warren Harrison

This thesis presents a new, practical approach to solve various NP-hard combinatorial problems of logic synthesis, logic programming, graph theory and related areas. A problem to be solved is polynomially time reduced to one of several generic combinatorial problems which can be expressed in the form of the Generalized Propositional Formula (GPF): a Boolean

product of clauses, where each clause is a sum of products of negated or non-negated literals.

The special parallel computer architecture for solving the GPF minimization problem, called the Generalized Propositional Formula Solver (GPFS), is discussed. The GPFS is composed of a Host computer and a convergent Data Flow Tree (DFT) of Boolean Product Processors (BPP) tightly coupled with it. Each BPP consists of a Product Management Unit (PMU) and a systolic Sorting and Absorbing Parallel Architecture (SAPA).

A simulation program was developed to verify the correctness of the GPFS algorithm, to examine the efficiency of the intercommunications between the processors at different levels of the data flow tree, and to optimize the processor utilization. The test results reveal that for practical applications, the GPFS, implemented with three BPPs, can be used as a special accelerator to solve GPF optimization problems of 1000 clauses or less, at the rate of 100,000 to 1,000,000 times faster than to solve the same problems on a conventional computer with the same operating speed.

The Covering Problems Solver (CPS) algorithm is included as an extension to the GPFS. This program performs Quine-McCluskey method to reduce the size of the switch table, and combines the implicant domination properties with the breadth first tree search (BFS) to efficiently simplify and accelerate the process of finding the minimal cover sets. A table of the test results is also provided.

PARALLEL ARCHITECTURES

FOR SOLVING COMBINATORIAL PROBLEMS OF LOGIC DESIGN

by

PHUONG MINH HO

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University

1989

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Phuong Minh Ho presented August 7, 1989.

Marek Perkowski, Chairman

Michael Driscoll

Robert Daasch

Warren Harrison

APPROVED:

Rolf Schaumann, Chair, Department of Electrical Engineering

C. William Savery, Vice Provost for Graduate Studies and Research

ACKNOWLEDGMENTS

I wish to express my special thanks to Dr. Marek Perkowski, my thesis advisor, who has guided me toward the accomplishment of this thesis with many insightful comments, helpful suggestions, and many long hours discussing various approaches and ideas about the presented architectures.

I also wish to give thanks to my grandmother and my parents for their encouragements.

Lastly, my sincere gratitude is due to my wife, Hoai-Thu, and my girl, Bich-Ngoc, for their love, patience, and inspiration.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In spite of many advances in parallel processing architecture concepts and breakthroughs in VLSI technology in the last few decades, many classic computational problems still cannot be effectively resolved by our conventional general purpose computers due to the time complexities of the applied algorithms.

THE NP COMBINATORIAL PROBLEMS

According to Garey and Johnson [1979], "the time complexity function for an algorithm expresses its time requirements by giving, for each possible input length, the largest amount of time needed by the algorithm to solve a problem instance of that size". A growing list of more than 300 intractable problems in the fields of Graph Theory, Network Design, Set and Partitions, Storage and Retrieval, Sequencing and Scheduling, Mathematical Programming, Algebra and Number Theory, Games and Puzzles, Logic design, Automata and Languages, Program Optimization, and many other miscellaneous problems of the same class can be found in this source. Since the intractability of these problems adheres to the fact that no polynomial time algorithms have been

feasibly devised to effectively solve them, theoreticians have proved and classified these intractable problems as NP problems. Polynomial time algorithms are those which have time complexity function of O(p(N)), where N is the number of variables in the problem and p(N) is a polynomial function of N. The name "Nondeterministic Polynomial" (NP) arises from the fact that the problems in this class can be solved in polynomial time by a theoretical (abstract model) nondeterministic computer which can perform an infinite number of computations independently, in parallel. The time complexity of the known, straightforward step-by-step procedures which can be applied for solving these problems on a conventional computer is defined to be of O(f(N)), where f(N) is an exponential function of the problem size. On the other hand, there are many problems which can be solved in polynomial time by a deterministic computer. These problems are classified as P problems. A deterministic computer is a computing model which has its next state to be determined uniquely by the present state and the input control signal, with respect to a state transition function [Kohavi, 1978], [Sedgewick, 1983]. All NP (decision) problems inherit the characteristic that if the answer to the given problem instance is "YES", its correctness may be verified in polynomial time. However, if the obtained answer is "NO", there is no better way to perform this verification process other than to arbitrarily guess (nondeterministically) all

possibilities and verify the validity of each trial. If a problem, belonging to the P class, can be solved by a deterministic polynomial time algorithm, then the complementary problem (also belonging to P) can be solved as well. However, the same principle may not be applied for the problems in NP class. It is believed that P is a proper subset of NP. Yet, no proof or disproof has been submitted. [Garey and Johnson, 1979], [Horowitz and Sahni, 1978], [Hu, 1982], [Page and Wilson, 1979].

Since the exponential function grows drastically faster than the polynomial function with respect to the increasing exponent values (which is the size of the problems in this case), the exponential time algorithms are regarded as inefficient algorithms in comparison with their good or efficient counterparts, the polynomial time algorithms. This point is clearly illustrated [Garey and Johnson, 1979] in comparing a set of polynomial and exponential time complexity functions: an N^3 algorithm requires 0.001 second, 0.027 second, and 0.216 second to solve problems of sizes N = 10, 30, and 60, respectively; while a 3^N algorithm will require 0.059 second, 6.5 years, and 13 trillion centuries to solve the same set of problems. Garey and Johnson have shown that the expansion in size of the problems which must be solved by an exponential time algorithm, in a specific length of time, is insignificant with respect to the increases in operating speeds of the utilized computer systems. For example, if a

current technology computer can solve a problem instance of size N in an hour utilizing an algorithm of time complexity N^3, a computer 100 (or 1000) times faster can solve a problem of size 4.64N (or 10N), in the same length of time, by applying the same algorithm. On the other hand, if a present technology computer can solve a problem of size M in one hour, using an algorithm of time complexity 3^M, a computer 100 (or 1000) times faster can solve a problem of size M + 4.19 (or M + 6.29, respectively) in the same period, with the same algorithm.

Computer scientists have shown that all NP problems can be reduced to a single decision problem (i.e., Yes/No problem) known as the Satisfiability Problem (NP-complete), and that the time to perform this reduction (also called transformation) can be expressed in a polynomial expression [Cook, 1971], [Karp, 1972], [Ralston and Reilly, 1983], [Horowitz and Sahni, 1978], [Hu, 1982]. Thus, if a polynomial time algorithm can be found for the Satisfiability Problem, then efficient polynomial time algorithms can be devised for all NP-complete problems. The "polynomial time reducibility" is of fundamental importance in the theory of NP-completeness and combinatorial algorithms.

POLYNOMIAL TIME REDUCIBILITY

As Cook's theorem states - "Satisfiability is NP-complete"; problems which can be reduced to this "first" NP-

complete problem belong to the set of NP-complete problems. Since the list of NP-complete problems grows with time, the task of proving that a combinatorial problem is NP-complete is consequently getting easier: simply prove that the problem instance of interest is polynomially time reducible to a known NP-complete problem. Besides the NP-complete problems, there exist many other decision and optimization problems, in different disciplines, which are categorized as NP-hard problems: "Any decision problem, whether a member of NP or not, to which we can transform an NP-complete problem will have the property that it can be solved in polynomial time unless P = NP" [Garey and Johnson, 1979]. These NP-hard problems share a common property that if they can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time as well [Hu, 1982].

The idea of polynomial time reducibility in NP problems suggests that if we have a hypothetical polynomial time algorithm for the satisfiability problem, then this algorithm can be used as a universal subroutine to solve all reduced instances of all NP-complete problems. More generally, as many practical (exponential time) algorithms have been developed for a number of combinatorial problems, a new NP problem can be transformed to a known problem and be solved by using the existing subroutine. Figure 1 depicts an example of polynomial time reducibility of a Graph theory problem to another of Set theory: reducing a Clique Decision problem to

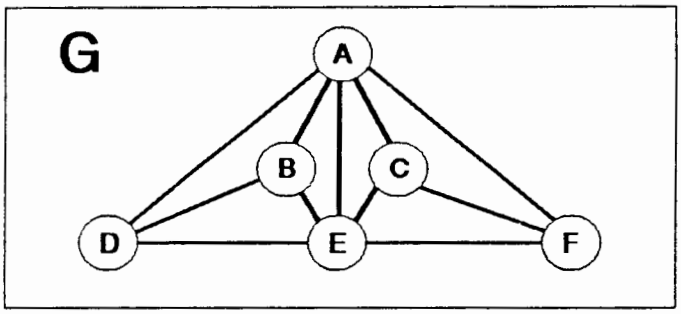
a Set Covering Decision problem [Garey and Johnson, 1979], [Horowitz and Sahni, 1978], [Ralston and Reilly, 1983].

In the Clique Decision problem, we are given a graph G = (V, E), where V and E are the sets of vertices and edges, and a positive integer K that is equal to or less than the number of vertices in the given graph. The task is to determine if G contains a clique Q of size K or more. Where Q is a subset of V, with a number of vertices less than or equal to K, and with every two of its vertices being connected by an edge.

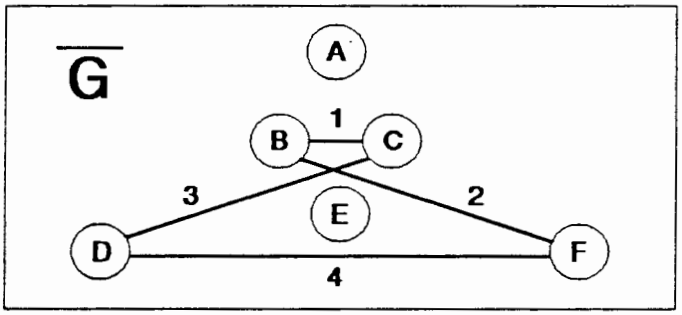
In the Set Covering Decision problem, we are given a collection U of subsets of a finite set S and a positive integer K with value less than or equal to the total number of subsets in the collection U. Does U contains a cover C for S (of size K or less) such that every member of S belongs to at least one member of U ? Where C is a subset of U, and the number of subsets in U is less than or equal to K such that the union of all subsets in U includes all members of S.

In this example, the given graph G contains six vertices and two cliques of size four (ABDE and ACEF). Hence, for the clique problem, if K is set to four or less, the answer to this problem would be "YES". Otherwise, it would be "NO". The corresponding Set Covering Decision problem has four elements in the set S to be covered. Four is the number of edges in G', the complementary graph of G. S = {1,2,3,4}. The cover size K is two, which is the difference between the

number of vertices and the size of the clique. The solutions to the sample problem are either collections of the subsets (S_B, S_D), or (S_C, S_F). Thus, a "YES" answer is received if the covering size to this problem is set to 2 or more. Otherwise, the answer would be "NO".



Number of Vertices = 6
Max Clique = 4
Covering Size = 6 - 4 = 2



$S = \{ 1, 2, 3, 4 \}$

S_A	$= \{ \}$	S_D	$= \{ 3, 4 \}$
S_B	$= \{ 1, 2 \}$	S_E	$= \{ \}$
S_C	$= \{ 1, 3 \}$	S_F	$= \{ 2, 4 \}$

Figure 1. Reduction from clique decision problem to set covering decision problem.

The corresponding algorithm to transform the Clique Decision problem to the Set Covering Decision problem, as illustrated in Figure 1, consists of four main steps:

(1) Calculate the cover size of the Set Covering Decision problem.

(2) Construct a complementary graph of the given graph G by forming a graph G' which possesses the same number of vertices as G does, but contains edges between any pair of vertices that are not connected in the original graph.

(3) Label the edges and form the set S of all edges in G'. This is the set to be covered.

(4) Construct the collection, U, of N covering subsets at each vertex of the complementary graph. These subsets contain respectively the labeled edges incident to a particular vertex of G'. For a graph of size N, there shall be N subsets, and the number of elements in each subsets may be between zero and N - 1.

Suppose that we already possess a practically good algorithm to solve the Set Covering Decision problem. Upon confrontation with a Clique Decision problem , we can employ this procedure to reduce the Clique Decision problem to the Set Covering Decision problem, then invoke the existing subroutine to solve the reduced problem. The answer to the reduced problem is also the answer to the original problem. The above transformation procedure is easily seen to be a polynomial time algorithm since each step in it can be

executed in O(N) execution cycles.

The Set Covering Decision problem is very similar to another problem in Graph theory: the Vertex Cover Decision problem, in which we are given a graph G (V, E) of N vertices and M edges, and a positive integer K, where K <= N, and are asked to determine whether or not there exists a vertex cover of size K or less for the graph G. A vertex cover is defined as a subset V' (of vertices) of the set V, such that there is at least one vertex of each edge in the given graph belonging to V'. The Clique Decision problem can also be reduced to a Vertex Cover Decision problem. If G is the given graph with N vertices and K is the number of clique size in G, by the definition of clique, all these K vertices are connected by edges. Therefore, no edge exists between these vertices in the complementary graph G'. The set of edges in G' must be formed by connecting the remaining (N - K) vertices, which are not connected in G. Analogously, if V' is the vertex cover of G' then (V - V') must form a complete subgraph in G.

It can be shown that the polynomial time reducibility property is not exclusively restricted to decision problems in NP class, but can be applied to optimization problems as well. Let us consider the transformation of the Maximum Clique Optimization problem to the Clique Decision problem. The Maximum Clique Optimization problem requires us to find the largest possible clique size in a given graph G containing N vertices and M edges. If we have a deterministic polynomial

algorithm for the Clique Decision problem, then just by repeatedly calling this subroutine N times (at most) to solve the Clique Decision problem, and decrementing the clique size K (initialized to N) at each invocation, we can obtain the solution to the Maximum Clique Optimization problem in at most N times (worst case) of the subroutine execution time to solve the Clique Decision problem. Obviously, the above process should be terminated immediately after the first "YES" answer to the Clique Decision problem is obtained. This fact clearly indicates that the algorithm for solving the NP optimization problem must have the time complexity at least the same as that of the corresponding decision problem. Therefore, it can be solved in polynomial time if and only if the corresponding decision problem can be solved in polynomial time [Hu, 1982].

Tremendous efforts have been devoted in searching for polynomial time algorithms for these NP-complete problems; however, no successful results have yet been announced. In the meantime, before a "major intellectual breakthrough" [Ralston and Reilly, 1983] arrives, some approaches may be employed to obtain practical solutions for these inherently intractable problems:

(1) It is possible to design fast algorithms to solve some narrower special cases of combinatorial problems or to design algorithms that are particularly efficient to resolve problems of small sizes. Moreover, if a problem appears to inherit certain special characteristics, a

special algorithm may be designed to quickly solve it by reducing the dimension of the problem based on these special characteristics.

(2) Many practical combinatorial algorithms have been developed [Hu, 1982], [Bose and Manvel, 1984], [McVitie and Wilson, 1971]. Although these algorithms have exponential time complexity and may not guarantee absolute optimal solutions for all problem instances in all cases, they are reasonably adequate for many realistic applications.

(3) For large size optimization problems, instead of aiming at exact optimal solutions, it is pragmatically acceptable to find quasi-optimal solutions (in a realistic length of time) by heuristically applying problem decomposition, partitioning methods, and tree searching techniques [Perkowski and Brandenburg, 1989] to the original problem to break it into sub-problems of smaller, solvable sizes. Then fast algorithms can be developed to solve these sub-problems separately. The combinations of these partial results are the final solutions for the given problem.

(4) Along with these software-oriented approaches, the design ideas for special purpose computing systems for solving combinatorial problems have gradually been emerging into reality. These hardware accelerators utilize many parallel processing architectural concepts

and most of the algorithmic design methodology mentioned above. These designs are the main focus of this thesis.

ORGANIZATION OF THIS THESIS

This section provides a brief outline of the subsequent four chapters in this thesis.

In Chapter II, a brief discussion of the NP-hard problems in the field of Logic design, and the reducibility among these problems into some generic problems are presented. The Generalized Propositional Formula (GPF) will be introduced, together with its applications, along with the concept of Logic Design Machines.

Chapter III presents the Generalized Propositional Formula Solver (GPFS). The detailed architectural and operational description of the design is provided.

Chapter IV is devoted to the performance analysis of the GPFS designs. The description of the GPFS simulator and the simulation results are presented. Evaluation for the GPFS architecture in various design aspects, such as structural regularity, modularity, programmability, flexibility, versatility, performance efficiency, realization feasibility, cost effectiveness, etc., are presented. The GPFS architecture, with its high versatility and simple structural processing elements, is shown suitable for practical implementation.

Chapter V contains some possible extensions to improve the performance of the GPFS architecture. The Covering Problem Solver (CPS) algorithm is introduced as a possible extension to the GPFS to obtain an optimal subset of implicants which cover the entire set of literals in the original GPF. This chapter closes the thesis with some concluding remarks.

Appendix A contains the detailed architectural and operational descriptions of the Sorting and Absorbing Parallel Architecture (SAPA). Appendices B and C respectively include the source code listing of the GPFS simulation program, test cases, and test results for the GPFS architecture. The CPS program listing, test cases and test results can be found in Appendices D and E.

CHAPTER II

THE COMBINATORIAL PROBLEMS OF LOGIC DESIGN AND THE CONCEPT OF LOGIC DESIGN MACHINE

Most of the important combinatorial logic design problems such as Minimization of Programming Logic Array (PLA), PLA folding, State-assignment and Minimization of Finite State Machines (FSM), Testing, Fault-location, Multi-level Logic Design, etc., are NP-hard problems [Ho and Perkowski, 1989]. Let A, B, C,..., and A', B', C',..., respectively denote Boolean variables and their negations. It was observed that many logic design combinatorial problems can be initially expressed as expressions in either Conjunctive Normal Form (CNF), or Disjunctive Normal Form (DNF), which are Boolean functions in Product of Sums (POS), or Sum of Products (SOP) of literals, respectively. The sums (or products) in the POS (or SOP) expressions are called "clauses", and the literals in these clauses represent Boolean variables or their complements.

THE GENERIC COMBINATORIAL PROBLEMS OF LOGIC DESIGN

With the above definitions, many of the NP-hard logic design problems can be reduced to one or more generic combinatorial problems, listed as follows.

(1) Satisfiability: Given a CNF formula, determine if there exists a product of literals that satisfies all clauses. Answer "YES" if such a product exists; otherwise, answer "NO".

(2) Petrick Function Minimization: Given a CNF formula, find a product with a minimum number of variables that satisfies all of the clauses, or prove that such a product does not exist [Petrick, 1959].

(3) Optimization of Generalized Petrick Function: Given a CNF formula, find a product of literals that satisfies all of the clauses and has a minimum number of literals of non-negated variables, or prove that such a product does not exist [Perkowski, 1985].

(4) Partial Satisfiability: Given a CNF formula, find a product of literals that satisfies most of the clauses.

(5) Conversion from CNF to DNF: Given a CNF formula, convert it into a DNF formula (i.e., convert a Boolean function from Product of Sums to Sum of Products form).

(6) Conversion from DNF to CNF: Given a DNF formula, convert it into a CNF formula (i.e., convert a Boolean function from Sum of Products to Product of Sums form).

(7) Complementation of Boolean Function: Given a DNF formula, find its complement in the same form.

(8) Tautology Checking: Given a DNF formula, verify whether it is a Boolean tautology. In propositional calculus, a formula is defined as a tautology if and only if it is

true for all possible truth assignments to its variables. Non-Tautology, the complementary problem of this, is an NP-complete problem: Given a propositional formula, determine if there is a truth assignment for its variables that makes the given formula to be false (i.e., is the given formula not a tautology ?).

Being "Hard" problems in the NP class, these combinatorial optimization problems can be reduced to another problem in polynomial time. For instance, the Boolean Minimization problem can be reduced to the combination of the Set Covering and the Maximum Clique Optimization problems, or to the Graph Coloring problem. These problems can be further reduced to the Petrick Function Minimization (PFM) problem, with all literals in non-negated form. On the other hand, the State-assignment problem of FSM is reducible to the Multiple-valued Input Logic minimization problem. The FSM State Minimization problem can be transformed into the Covering/Closure problem. Both the Covering/Closure problem and the Petrick function minimization problem can be further reduced to the Generalized Petrick Function Optimization problem (GPFOP). The GPFOP can be reduced to the Satisfiability problem (SAT), and further to the Three-Satisfiability problem (3-SAT), where each clause contains exactly three literals [Perkowski, 1985], [Garey and Johnson, 1979]. An illustration of this polynomial time reducibility is shown in Figure 2.

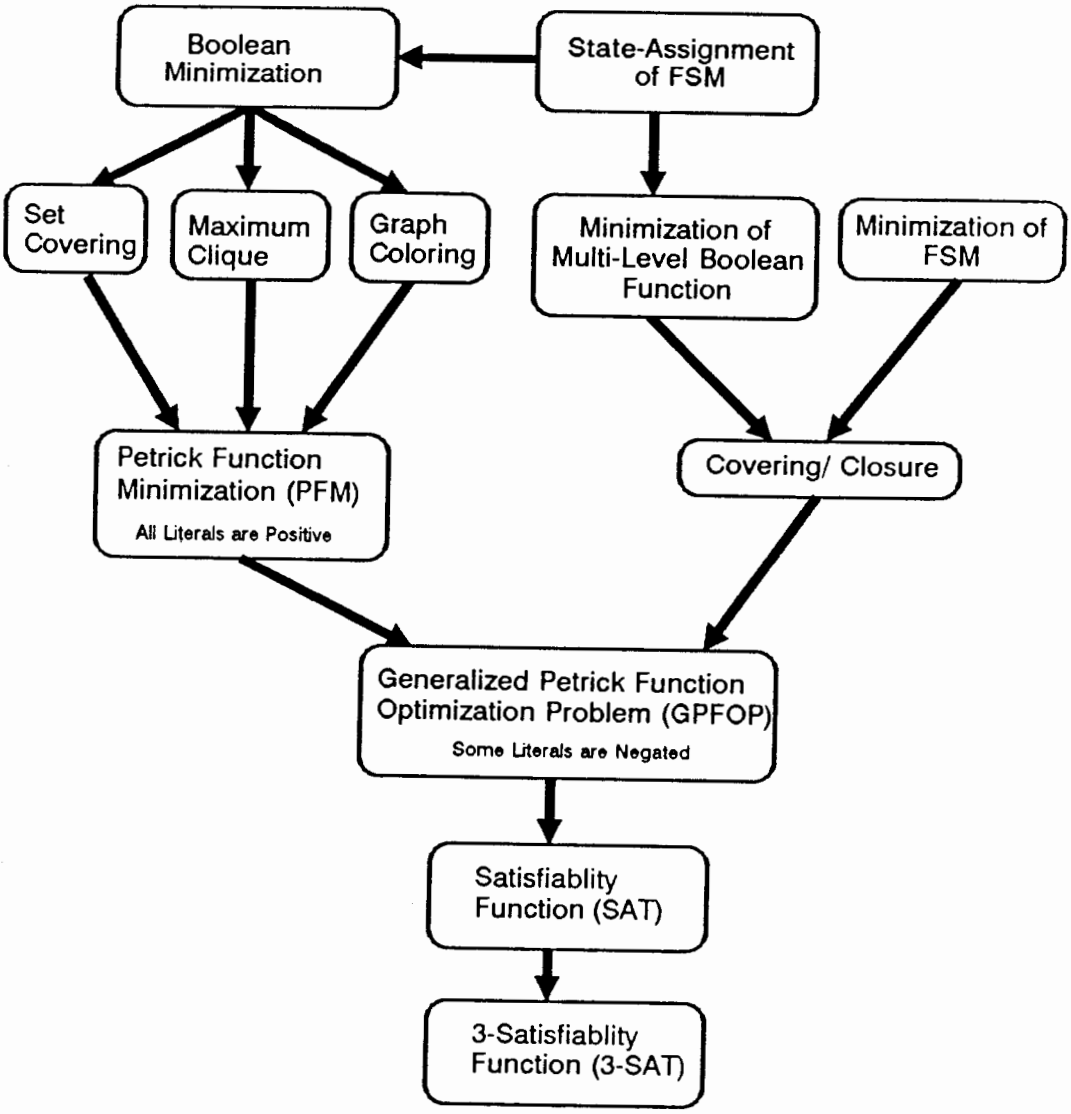


Figure 2. Polynomial time reducibility of NP-hard problems of logic design.

To further illustrate the above reductions, the following example represents the transformation of the Micro-instruction Minimization problem of micro-programming into the Set Covering Optimization problem (in tabular form). Given a set of micro-instructions (I_1, I_2, I_3, and I_4), each instruction

must be associated with a number of control signals. I_1 with {A, B, C, G}, I_2 with {A, C, E, H}, I_3 with {A, D, F}, and I_4 with {B, C, F}. The given information can be tabulated as shown in Figure 3a. The second step is to form all possible Maximum Compatibility Classes (MCC), which is the set S of all possible maximal combinations of different control lines that can be simultaneously asserted high to activate a group of micro-instructions in the set without encountering a conflict. The MCCs can be found by combining rows without conflicting column symbols. Since the table contains four columns, corresponding to the number of the micro-instructions in the set, four subsets of MCCs can be formed: S_i, where $1 <= i <= 4$, is the subset of MCCs, which are constructed by combining i rows in the table, under the condition that if an "X" appears in a column of a row, it may not be in the same column of the others. After that, all the MCCs are checked against others: If an MCC includes another MCC, it will be removed from the set S. (e.g., BD is included in BDE, and will be eliminated from the set of MCCs). These operations are presented in Figure 3b. In the third step, a table of all remaining MCCs versus control signals is constructed (Figure 3c). This table consists of 8 rows, corresponding to 8 remaining MCCs (C_1 = A, C_2 = CD, C_3 = BDE, C_4 = BDH, C_5 = DEG, C_6 = DGH, C_7 = EFG, C_8 = FGH); and 8 columns, corresponding to 8 signals (A, B, C, D, E, F, G, and H).

(a)

I_1 = A, B, C, G
I_2 = A, C, E, H
I_3 = A, D, F
I_4 = B, C, F

	I_1	I_2	I_3	I_4
A	X	X	X	
B	X			X
C	X	X		X
D			X	
E		X		
F			X	X
G	X			
H		X		

(b)

$S = S_1 + S_2 + S_3 + S_4$

S_1 = { A, B, C, D, E, F, G, H }

S_2 = { BD, BE, BH, CD, DE, DG, DH, EF, EG, FG, GH }

S_3 = { BDE, BDH, DEG, DGH, EFG, FGH }

S_4 = { }

(c)

MCCs	A	B	C	D	E	F	G	H
C_1 = A	X							
C_2 = CD			X	X				
C_3 = BDE		X		X	X			
C_4 = BDH		X		X				X
C_5 = DEG				X	X		X	
C_6 = DGH				X			X	X
C_7 = EFG					X	X	X	
C_8 = FGH						X	X	X

(d)

MCCs	B	E	F	H
C_3 = BDE	X	X		
C_4 = BDH	X			X
C_7 = EFG		X	X	
C_8 = FGH			X	X

Figure 3. The transformation of the micro-instruction minimization problem into a table covering problem. (a) Micro-instruction table. (b) Maximum compatibility classes. (c) MCC table. (d) MCC reduced table.

The table is further simplified by:

(1) Removing all essential MCCs and the associated rows and columns. Essential MCCs are the only contributors to a column in the MCC table. They must be included in the final solutions. In this example, C_1 = A and C_2 = CD are two essential MCCs to be removed, and in the result, columns A, C, and D are eliminated.

(2) Removing all dominating columns. A column is said to dominate another if the first column has an "X" in every row in which the second column has an "X". Thus, as G dominates F, it is excluded from the table since a minimal cover derived from the table with the existence of both columns can be derived from the table with only the dominated column.

(3) Removing all dominated rows. A row is dominated by another if it covers all the literals that the other does. Correspondingly, C_5 and C_6 are eliminated since they are dominated by C_7 and C_8, separately.

The three operations described above shall be repeated until no essential MCCs, dominating columns, and dominated rows can be detected. At this stage, the original problem is transformed into a table covering problem, where we need to find a minimum number of MCCs to cover the reduced table. Since the dimension of the reduced table is relatively small in this example (Figure 3d), it can be easily observed that the table is covered by either $\{C_3, C_8\}$, or $\{C_4, C_7\}$. To

verify this solution, let us reexamine the original MCC table. Since literal G dominates F, and A, C, D are covered by the essential MCCs C_1 and C_2, the table contains four remaining columns: literal B is covered by C_3 and C_4; E is covered by C_3, C_5, and C_7; F is covered by C_7 and C_8; and H is covered by C_4, C_6, and C_8. The corresponding Boolean POS expression is as follows:

$$(C_3 + C_4)\ (C_3 + C_5 + C_7)\ (C_7 + C_8)\ (C_4 + C_6 + C_8) = 1$$

Which can be expanded to

$$(C_3 + C_4C_5 + C_4C_7)\ (C_8 + C_4C_7 + C_6C_7) = 1$$

$$C_3C_8 + C_4C_7 + C_4C_5C_8 + C_3C_6C_7 = 1$$

The smallest products are C_3C_8 and C_4C_7, which cover the reduced table. The final solution to this problem can be either (C_1, C_2, C_3, C_8) or (C_1, C_2, C_4, C_7). Both solutions are equally optimized. Similarly, many other problems in Switching and Finite Automata Theory can be transformed to this Set Covering Optimization problem, using the same method of table reduction [Kohavi, 1978], [Hayes, 1978].

Since all NP-complete combinatorial decision problems are polynomially time reducible to a Satisfiability problem, we will introduce a Boolean expression in a more general form which can be used to express this type of problem. We also suggest a name for it: The Generalized Propositional Formula (GPF). The GPF is expressed in the form of product of

clauses, where each clause is a Sum of Products of literals, and the literals in the products may be in a negated form. An example of a GPF is as follows:

(ABC' +..+ G'K) (CDF' + XZL' + PMH +..) .. (MAP' +..+ W)

The GPF find application in Petri net analysis, expert systems, operations research, graph theory, search theory, cryptography, and many problems of logic and VLSI design. Let us reconsider the problem in Figure 1, and assume that the maximum clique size of the graph G is to be found. From the complementary graph G', the problem can be formulated as in the following CNF expression,

(B + C) (B + F) (C + D) (D + F)

which is a simplified GPF. This GPF is further expanded into a Boolean sum of products form. Then by excluding the literals of the minimum products from the set of all literals (which correspond to the vertices on the given graph), the maximum cliques can be found. Thus, the above expression can be converted to BD + CF. Two maximum cliques, both of size 4, are detected: ACEF (i.e., ABCDEF - BD), and ABDE, (i.e., ABCDEF - CF).

Garey and Johnson have shown that every combinatorial problem in their extensive collection of NP-hard and NP-complete problems [1979] can be reduced to a GPF minimization problem. Although in some cases, the transformations produce

problems of large sizes and seem to be suitable only for theoretical experiments, the reductions are practically reasonable for many problems. Furthermore, by analogy, one might suspect that new NP-complete problems to be added to the list will be reducible to GPF minimization problems as well.

The algorithms for solving the GPF can be classified into three categories: Tree searching algorithms, array algorithms, and transformational algorithms. All of these algorithms can be sequential or parallel algorithms, and they can be executed in either standard or special architectures. In the scope of this thesis, the first two types are assumed to be executed by the Host computer, and the third one by the Generalized Propositional Formula Solver (GPFS), the parallel architecture discussed in Chapter III.

THE PARALLEL LOGIC DESIGN MACHINE CONCEPT

It is not well known that logic computers, which are devices to aid in verifying syllogisms and solving other logic problems, are older than the digital arithmetic computers and date back to the Middle Age "computers" by Lullus. A variety of such computers were built in the XIX and XX centuries before VonNeumann. The first, to our knowledge, special computer to help in logic synthesis was proposed by Antonin Svoboda [Svoboda, 1973] in Czechoslovakia, and then in the United States. Since the hardware accelerators for simulation, design rule checking, routing, placement, and other layout

tasks are now available or proposed, we believe that hardware accelerators for logic design will also be incorporated into future CAE workstations for VLSI design because there is an obvious and growing need for them.

CHAPTER III

THE GENERALIZED PROPOSITIONAL FORMULA SOLVER

This chapter introduces the Generalized Propositional Formula Solver (GPFS), a parallel architecture for solving various NP-hard problems in logic synthesis, logic programming, graph theory, and related areas. As discussed in Chapter II, most of these combinatorial problems can be reduced to one or several generic decision or optimization problems that can be formulated into a generic Boolean expression: the Generalized Propositional Formula (GPF). The GPF is expressed in the form of product of clauses, where each clause is a Boolean sum of products of literals. The literals in these products may be in either negated or non-negated form. For example, the AND-OR tree search problem, minimization of output phase of Programming Logic Array [Sasao, 1984], minimization of Boolean relations [Brayton, 1989], covering/closure problems [Kohavi, 1970], all can be expressed in the GPF form. The requirement for these optimization problems is to find a minimum subset of arbitrary literals which satisfies the given GPF. The GPFS is designed to achieve this goal.

ARCHITECTURAL DESCRIPTION

Assumptions have been made in the conceptual design of the GPFS architecture. Practically, the problems related to Boolean minimization often require a relatively small number of variables. Thus, a computer word of 256 bits is sufficient to represent a Boolean product of up to 124 literals, by using two bits to encode a literal (since the GPF allows negated literals). The storage organization of each 256 bit word, as shown in Figure 4, is partitioned into three parts, described as follows.

(1) The Tag Field: the most significant bit (location 255) of the 256-bit word is reserved for the Tag Field. This tag bit, if reset to 0, indicates that the associated product is a dominating product (which dominates at least one other product in the same clause). Dominating products are rejected at the output phase of the SAPA operations.

(2) The Cost Field: the next 7 bits (locations 254 to 247) constitute the Cost Field. This field contains the number of existing literals (01, 10) in a product. The product cost value is in the range of 0 and 127.

(3) The Product Field: the remaining 248 least significant bits (locations 0 to 247) of the 256-bit word form the Product Field. According to the GPFS encoding scheme, shown in Table I [Dietmayer, 1971], the bit patterns 01

and 10 respectively represent literals in negated and non-negated forms. The logical value 00 indicates a contradiction, which is the result of logically ANDing a Boolean variable with its complement. And the value 11 represents a Don't Care, which indicates that the corresponding literal is excluded from the encoded product. According to the number of bits in the product field and to the encoding scheme, the maximum number of literals (i.e., product cost) allowed in an encoded product is 124.

A product containing Don't Care (11) in all bit locations of the Product Field is called an "Empty Product". Empty products are used to initialize the system, and to separate clauses (i.e., clause delimiters) during data transfer between the Host and the GPFS processing units (called Boolean Product Processors), or between the Boolean Product Processors themselves. In contrast, a product containing Contradiction (00) for any literal in the Product Field is called a "Contradictory Product". Contradictory products are eliminated from the SOP expressions generated from each stage of the GPFS.

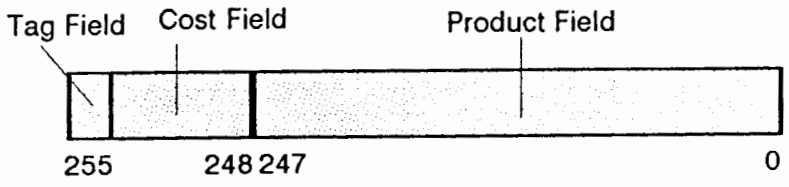


Figure 4. The GPFS word format.

TABLE I

THE GPFS ENCODING SCHEME

Codes	Symbols	Descriptions
00	Contradiction	Literal resulted from ANDing a literal with its complement
01	A'	Negated literal
10	A	Non-negated literal
11	Don't Care	Literal excluded from the encoded product

In this encoding scheme, the product of two products of literals is simply the result of a bitwise ANDing operation performed on the two 256-bit words [Brayton, Hachtel, McMullen, and Sangiovanni-Vincentelli, 1984]. For instance, with 5 variables, (A,B,C,D,E), the product of two given products (BE and BCD') is

```
BE . BCD'  = [11 10 11 11 10] . [11 10 10 01 11]
           = [11 10 10 01 10]
           = BCD'E
```

Whenever a Boolean literal is multiplied with its complement in another product, the bit pattern 00 is created from the bitwise ANDing operation. The resultant contradictory products are detected and rejected in the succeeding stages:

AB . AB'E = [11 10 11 11 11] . [10 01 11 11 10]

= [10 00 11 11 10]

= Contradictory Product

As depicted in Figure 5, the GPFS architecture consists of the Host, and tightly coupled with it - a convergent, balanced Data Flow Tree (DFT) of Boolean Product Processors (BPP). Each BPP is composed of a Product Management Units (PMU) and a Sum of Products Reduction Unit (SPRU).

The number of levels in the DFT may be arbitrary, and the application algorithms executed by the BPPs will not be affected with respect to the changes in number of levels of the DFT. The BPPs operate asynchronously, and may coordinate their processes with the Host (e.g., leaf node BPPs, or root node BPP), or with the predecessors and successors directly by some simple handshaking protocols. During the inter-levels communication process, possible data collisions may be avoided by means of software semaphores and arbiters. In the GPFS system, the data streams always flow from the Host to the leaf node BPPs, from the BPPs at higher level (called parent nodes, or predecessors) to BPPs at lower levels (child nodes, or successors), toward the root of the convergent DFT, and from the root node BPP back to the Host. The leaf node BPPs, with limited memory capacity, only request the Host to supply them with additional data as the numbers of clauses in their local memory drop down below a predetermined threshold value

which can simply be the upper bound of their memory capacity. The root node processor returns the partial results back to the Host whenever the number of clauses contained in its local

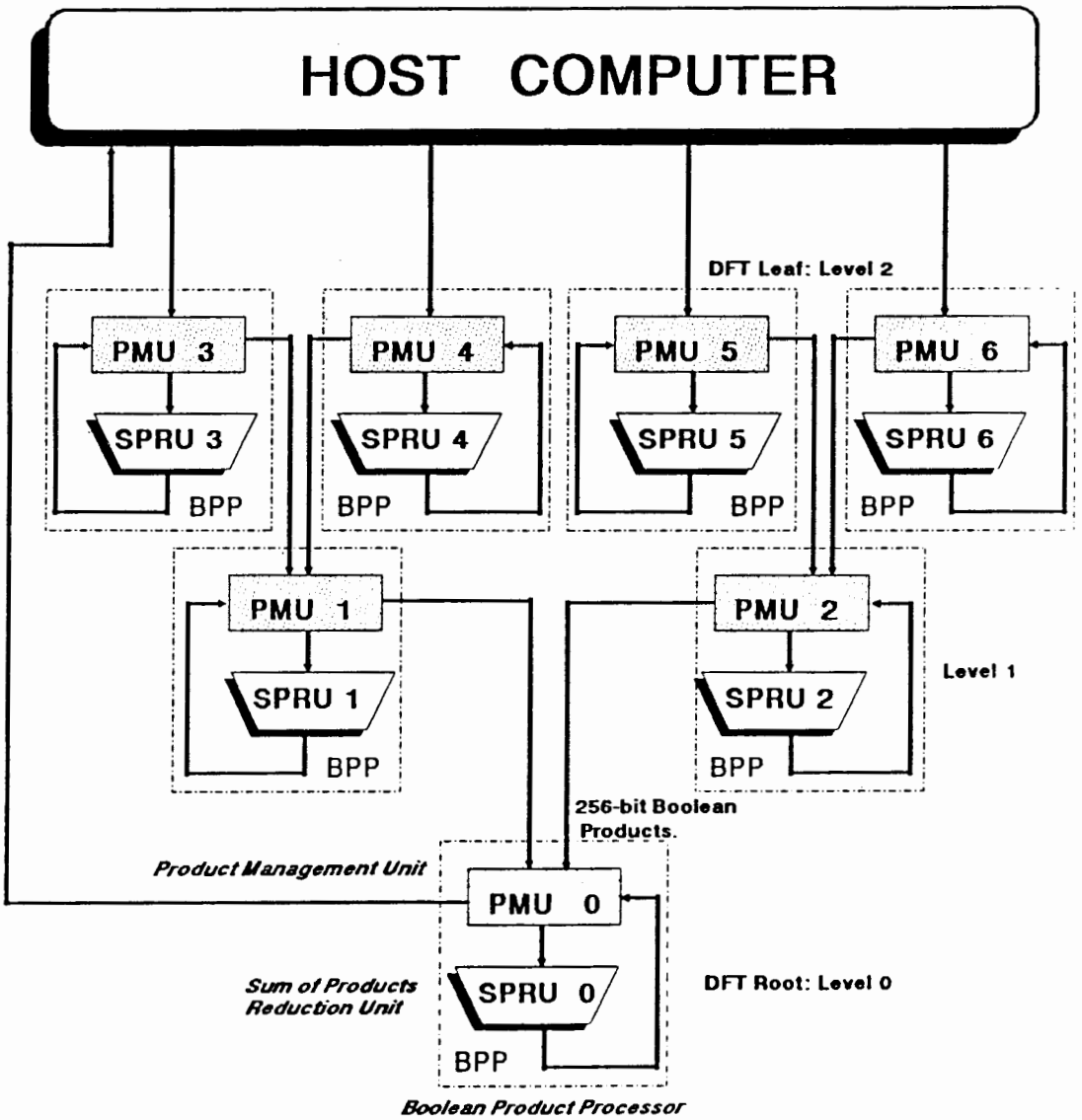


Figure 5. The GPFS architecture.

memory exceeds its memory capacity. In this operating fashion, the Host and the DFT of BPPs, together, constitute a closed loop of data transfer. Thus, with certain restrictions applied to the size of the memory storage in the BPPs of the DFT, the excessive data in the DFT will be sent back to the Host (through the root node) to be sent to the BPPs at the leaf-level of the DFT. Since there are more BPPs in the leaf-level, the entire process will be accelerated, and the system resources will be utilized more efficiently. As a design functional specification, the tasks performed by the Host, and by the Boolean Product Processors (BPPs) are listed in the following sections.

The procedure executed by the Host can be summarized in the following four major tasks:

(1) Decomposing the problem to fit it into the size constraints of the DFT by using tree search or partitioning methods.

(2) Fetching the leaf node processors of the DFT.

(3) Collecting the results in Boolean SOP expressions from the BPP at the root of the DFT.

(4) Determining the final solutions.

Concurrently, the Boolean Product Processors perform the following four procedures asynchronously, in parallel and pipeline fashion.

(1) Receive decomposed GPF from the Host (into the leaf node processors).

(2) Perform the Boolean multiplication on any two given clauses to generate all possible products. The given POS expressions are now replaced by SOP expressions.

(3) Calculate the costs of the products. Examine the bit patterns in the Product Field to detect and delete contradictory products.

(4) Sort the remaining products into descending order. Remove all dominating and redundant products. Select the first K most optimal products (i.e., the least cost products in the SPRU output sequences) and transfer them to the BPPs at the next level of the DFT.

The Product Management Unit (PMU)

The internal organization of the PMU - as illustrated in Figure 6 - includes a Control Unit (CU), a Cartesian Product Generator (CPG), and a local memory storage. This local storage must be sufficiently large to accommodate the SOP expressions accumulated from iteratively multiplying out the given clauses.

The Control Unit is responsible for I/O interfacing with the associated SPRU and its predecessor and successor nodes. The dynamic memory managing capabilities allow it to allocate/deallocate storage in the local memory unit to store new input data elements, and remove all existing items not further in use. The Control Unit also controls the Cartesian Product Generator operations, and directs all necessary local

and global communications in a BPP.

The CPG receives clauses (Boolean SOP expressions) from the PMU local memory into its internal register, R, and array of shift registers, AR, through the Control Unit. The principal function of the CPG is to perform the bitwise ANDing operation on every two 256-bit products of a pair of clauses to generate a new sequence of 256-bit Boolean products to be processed by the SPRU.

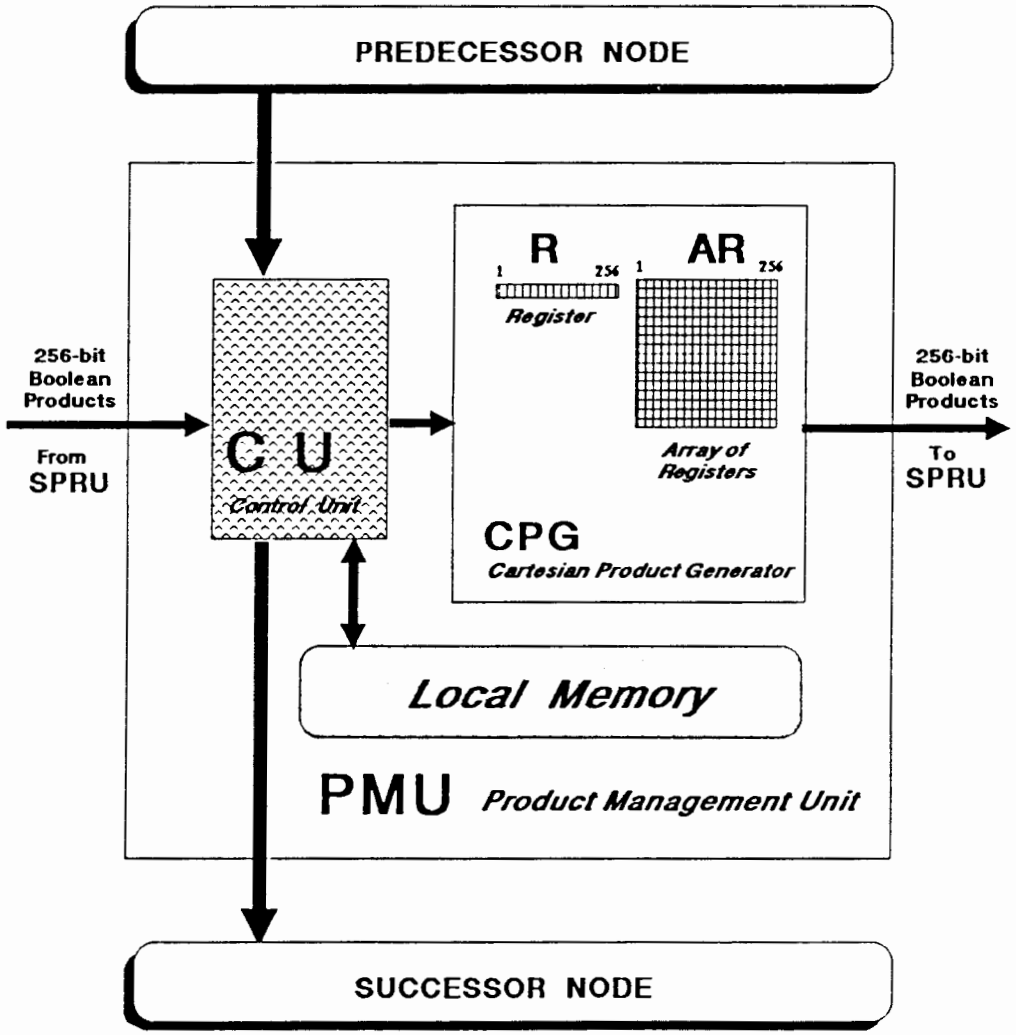


Figure 6. The product management unit organization.

At system initialization, all BPPs in the DFT send a signal to the processors to request initial data transfer. Accordingly, the Host fetches the clauses to the leaf node BPPs. As the process continues, the leaf node BPPs fulfill the requests of their successors as they completely multiply out the clauses received from the Host. The Control Unit in each PMU will allocate memory storage to accommodate these input clauses in the form of linked-list data structures. Data transmitted from the Host must be in the format of 256-bit word encoded products. Empty Products are used as delimiters to separate the clauses in the given GPF.

Upon completion of receiving initial data from the Host, the Control Unit in each BPP will retrieve a clause from local memory, load it into the array of registers, AR, of the CPG. As each product of the second clause is sequentially restored into register R in the CPG, it will be logically ANDed with every product of the first clause in AR. Local memory storage reserved for the retrieved data items is made available for later use. This memory management technique, known as dynamic storage allocation, is an essential capability for this application since the size of PMU local memory storage is limited, and the CPG constantly requires a large memory space to support its Boolean expansion process. As the CPG completely multiplies (i.e., logically ANDing) the product in R to every product in AR, the Control Unit fetches another product (of the same clause) into register R. When all

possible Boolean products, results of the multiplications of the second clause from local memory and the first clause in AR, have been generated, the Control Unit repeats the same operations with another pair of clauses. As this process continues, the newly generated products are sequentially transmitted to the accompanying SPRU to be sorted and eliminated (e.g., for cases of dominating and redundant products). During this process, if a request is pending, the expression arriving from the SPRU is delivered to the successor node. Otherwise, the SPRU output expression is transferred back to the PMU in the same BPP to be multiplied with another clause retrieved from local memory. The Control Unit must allocate memory to store these accumulated results in a new linked-list structure of product nodes in its local memory. The PMU receives the reduced clauses from the SPRU, repeats the multiplication and inter-level data communication processes, until only one clause remains in the local memory of the BPP at the root of the DFT. At this point, the root node BPP returns the last clause (i.e., the final solution) back to the Host, and signals the Host to terminate the process.

The Sum of Product Reduction Unit (SPRU)

The SPRU, consists of a Cost Evaluator and Detector (CECPD) and a pair of Sorting and Absorbing Parallel Architectures (SAPA) connected to the CECPD through a 1x2

multiplexer. Consequently, the output of these SAPAs must be demultiplexed prior to propagating to the BPP at the next DFT level. As illustrated in Figure 7, a SPRU receives Boolean products from the associated PMU into its CECPD. The CECPD computes the number of existing literals (01, 10) in the given 248-bit Product Field. The product cost is stored in the Cost Field. During the process of calculating the cost of a product, whenever a contradiction (i.e., 00 pattern) in the Product Field is detected, the CECPD will clear the bit in the associated Tag Field. Only products with the tag bit set to one are transmitted to the SAPAs. The SAPA detects and removes all dominating products from the received clause. The remaining products in each clause are sorted with respect to their costs. For a clause of N products, the SAPA requires 2N execution cycles to completely perform the above operations. Since the SAPA throughput is only half that of the CPG, two SAPAs are used in each SPRU. Suppose that there are four clauses in the PMU local memory: As the CPG multiplies out the first and second clauses, the resultant 256-bit Boolean products are sequentially loaded into the first SAPA through the CECPD; during the time this SAPA serially outputs the products in a descending order, the CPG multiplies out the third and fourth clauses and fills up the second SAPA. As this process continues, the SAPAs alternately generate ordered clauses and transfer them back to the PMU (through the SPRU's 2x1 demultiplexer). Further

detailed discussion regarding the SAPA architecture and its operations is provided in Appendix A.

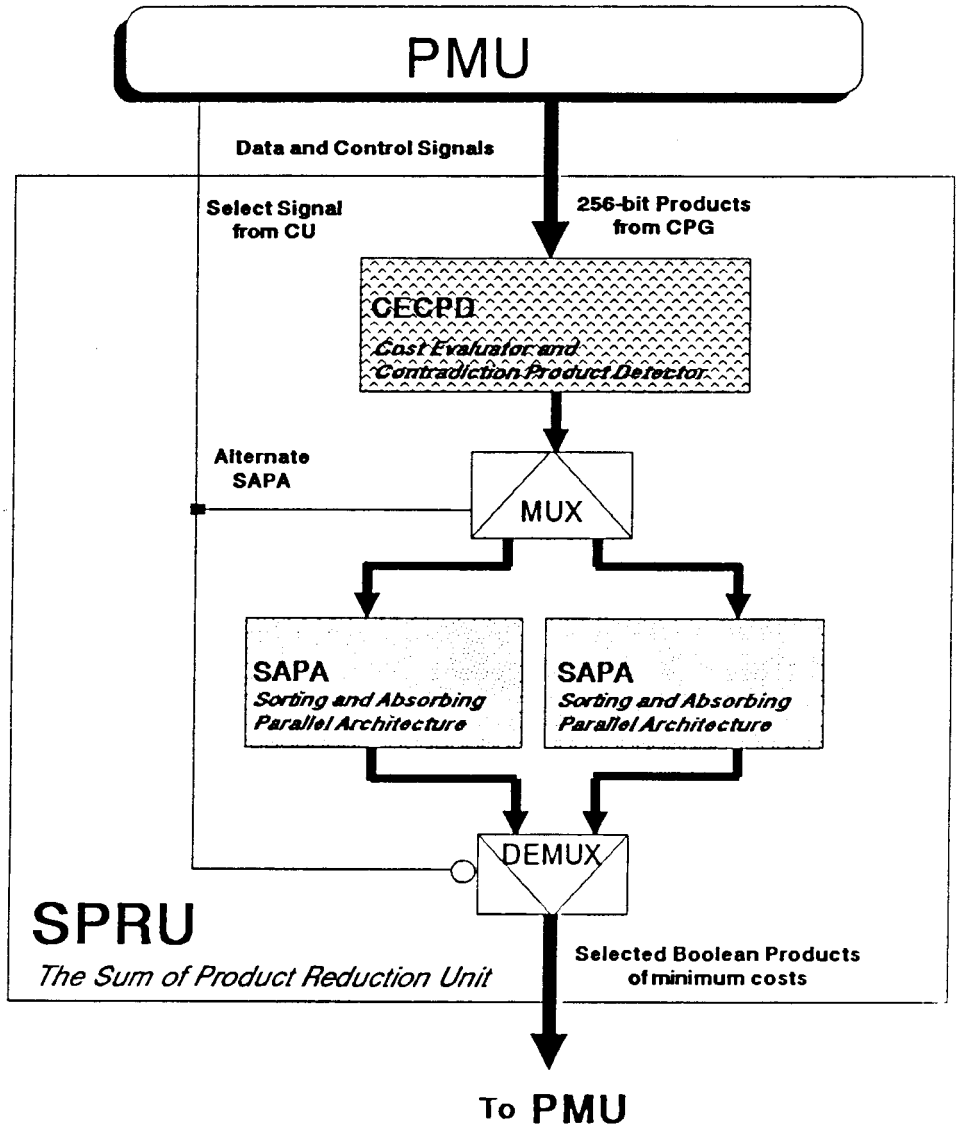


Figure 7. The sum of products reduction unit organization.

OPERATIONAL DESCRIPTION

The Host decomposes large problems to fit the constraints of the DFT, such as the word length, (i.e., number of literals), the size of memories in the BPPs, the number of clauses and products, and others. In the GPFS system - as depicted in Figure 5 - the Host loads the memories of the leaf node processors of the DFT whenever it acquires the "Send_Me" signal from the respective processor. In this configuration, the leaf node BPPs, residing at level 2, are labeled 3, 4, 5, and 6. If data is still available, the Host will set the "Connect" flag to indicate that a communication link to the requester has been established. Upon completion of transmitting a clause to the target BPP, the Host will clear the request with the "Clear" flag. If the Host has completed the data distribution to the leaf node BPPs of the DFT, the "Finish" flag will be set to signal the leaf node BPPs not to place further requests. All the BPPs in the systems, at different levels of the DFT, are coordinating their processes in a similar manner. The Host also receives the partial resultant SOP expressions from the root node BPP (i.e., BPP_0, at level 0). The GPF clauses are multiplied out and logically simplified as they traverse from the Host through the BPPs to the root of the DFT. From an external point of view, the entire process is analogous to the parallel realization of the simple Boolean multiplication and simplification algorithm for

a product of clauses: Each BPP in the DFT captures a part of the GPF (which can be treated as a smaller GPF), multiplies, and replaces it with a corresponding SOP expressions form, which becomes a part of a new GPF for the next phase of multiplication. The multiplications and replacements are done asynchronously, in parallel, and in a pipeline fashion inside the DFT. Each processor in the tree performs Boolean multiplication on the two input streams of data (Cartesian Product Generator, CPG).

The BPP simplifies the intermediate SOP expressions created by the CPG by removing all contradictory, duplicated, and empty products. For instance, if at anytime, the left and right SOP expressions clauses given to a BPP are AB and (A' + B') respectively, the multiplication result in this BPP, after the simplification, is an empty clause (i.e., all of its products are contradictory products). This information indicates that the total GPF has no solution. The Host may, consequently, generate a signal to clear all cells of the BPP processors, abort the entire operation, and start solving another GPF problem (if any).

Since contradictory and dominating products are removed from the SOP expressions, the sizes of the clauses do not grow too quickly when both negated and non-negated (i.e., positive) literals exist for the same variables in the given function. On the other hand, when all literals are positive as in the set covering problem, or when the GPF is a unate function

[Brayton, 1984], the clause sizes expand. In those cases, it can be expected that the clause size constraints of each processor are exceeded by the created products in its SOP expressions. However, if we are not interested in all products from the final SOP expressions, then with the sorting capabilities provided by the SAPA, only the best (least cost) products are selected from the resultant SOP expressions, and are transmitted to the higher level BPPs. Although this approach causes a loss of the optimal cover or not generating all implicants, the disadvantages are compensated by the gain in speed and the reduction in size of the required local memory. In addition, the simulation results proved that with reasonable memory storage in the BPP processors, the optimum solution is generated for the GPF problems in most cases.

In order to balance the processor loads and improve the data communication among the processors in the system, several approaches have been developed and implemented in the GPFS simulation program to verify the actual effectiveness of the system hardware utilization with respect to a number of parameters such as the problem size, the SAPA size, the number of DFT levels, etc. The algorithm executed by the Host may be summarized as follows.

The HOST Computer Algorithm

```
BEGIN
   (**   HOST: Get Input Data and Establish Database   **)
```

```
WHILE NOT EOF (Input_Data_File) DO BEGIN
   Literal := Get_a_GPF_Literal;
   IF (Literal = Product_Delimiter) THEN BEGIN
      Encode_the_Input_Product;
      Product_Count := Product_Count + 1;
      END (* IF *)
   ELSE IF (Literal = Clause_Delimiter) THEN
      HOST.Num_GPF_Clauses := HOST.Num_GPF_Clauses + 1;
   END; (* While: Input Phase *)

(**   HOST: Normal GPFS Operations  **)
WHILE NOT (GPFS_Finish) PARALLELLY DO BEGIN
   (**   HOST: Communicates with the root node BPP   **)
   IF (HOST.Mail = Connect) THEN BEGIN
      Returned_Product := Receive_a_Product_from_Root_BPP;
      IF (Returned_Product = Empty_Product) THEN BEGIN
         HOST.Num_GPF_Clauses := HOST.Num_GPF_Clauses + 1;
         HOST.Mail = Send_Me;
         END; (* receive a product from root BPP *)
      ELSE  Product_Count := Product_Count + 1;
      END  (* Host-Root communication link established *)
   ELSE IF (HOST.Mail = Clear) THEN HOST.Mail := Send_Me;

  (**   HOST: Communicate with the leaf node BPPs  **)
   FOR i:=1 TO Num_of_Leaf_BPPs DO BEGIN
      IF (GPFS[i].Mail = Send_Me) THEN BEGIN
         IF (HOST.Num_GPF_Clauses > 0) THEN BEGIN
```

```
                GPFS[i].Mail := Connect;
                Setup_to_Transfer_a_Clause_to_BPP;
                HOST.Num_GPF_Clauses:=HOST.Num_GPF_Clauses-1;
                END (* More clauses are still available *)
             ELSE  GPFS[i].Mail := Finish;
          END; (* Host acknowledges BPP requests *)
          ELSE IF (GPFS[i].Mail = Connect) THEN BEGIN
             IF (HOST.Num_Xfer_Products > 0) THEN BEGIN
                Transfer_a_Product_to_BPP;
                HOST.Num_Xfer_Products :=
                              HOST.Num_Xfer_Products - 1;
                END  (* send another product *)
             ELSE BEGIN  (* Completely transferred a clause *)
                Transfer_an_Empty_Product_to_BPP;
                GPFS[i].Mail := Clear;
                END; (* send an end-of-clause *)
             END; (* Host-BPP Data transfer in progress *)
          END; (* For: every BPP in the leaf level *)
       END; (* While: Normal GPFS Operation Phase *)

       Present_the_Final_Solutions; (* to the users *)
END. (** The Host Computer Algorithm **)
```

As shown in the above algorithm, the Host receives the given GPF, encodes the products into 256-bit words, keeps track of the number of products per clause and the number of clauses in the GPF, and stores the data in an array of doubled

linked-list structure. At every execution cycle, the Host checks the leaf node BPPs mail-boxes, and responds to them accordingly. During the same time fetching the leaf node BPPs, the Host may receive a partial resultant SOP expression returned from the root node processor (one product at every operating cycle). When the root node BPP returns the final clause back to the Host, it will set the "Finish" flag to TRUE: this will cause the Host to immediately conclude the normal GPFS operations, and to output the final solutions.

As the BPPs individually multiply together the clauses in their local memory storages, the number of 256-bit Boolean products will increase drastically. Even though the SPRUs have reduced the size of these expressions, the increase in number of newly generated Boolean products may quickly fill up the local memory space, and directly affect the execution time required by each BPP. At system initialization, the Host transfers blocks of data into the BPPs through the leaf node processors. After all BPPs received a sufficient number of clauses (two or more), the Cartesian Product Generating process may be activated. All BPPs at every level of the DFT concurrently process their clauses, reduce the size of the intermediate resultant SOP expressions by selecting only the most optimal products from the associated SAPA feedback streams, and accommodate these products in their local memory storages. These operations are done asynchronously and individually by each BPP. The GPFS algorithm is listed below.

The GPFS Algorithm

```
BEGIN
GPFS_Finish := FALSE;
Total_Execution_Cycles := 0;
WHILE NOT (GPFS_Finish) PARALLELLY DO BEGIN
   FOR I:= 1 TO NUMBER_OF_BPPS PARALLELLY DO BEGIN
      (** Step 1: Update System Status **)
      Update_PMU_Status;

      (** Step 2: Request Predecessors. Normal Execution **)
      Execute_the_Control_Unit_Algorithm;
      Execute_the_CPG_Algorithm;
      Execute_the_SPRU_Algorithm;

      (** Step 3: Transfer Data to Successor  **)
      Response_to_Successor_Requests;
      END;

   (** Root BPP transfers the final SOP expression **)
   IF ((GPFS[0].Num_Clauses = 1) AND
       (HOST.Num_GPF_Clauses = 0)) THEN BEGIN
      Transfer_a_Clause_to_Host;
      Termination_Status = NORMAL;
      GPFS_Finish := TRUE;
      END;
   END;
   Total_Execution_Cycles := Total_Execution_Cycles + 1;
END. (** The GPFS Algorithm **)
```

For the sake of completeness, the algorithms executed by the Control Unit, the Cartesian Product Generator, and the Sum of Product Reduction Unit in each Boolean Product Processor are presented in the following sections. These algorithms are invoked during Step 2 of the GPFS algorithm.

The Control Unit Algorithm

```
BEGIN
FOR I:=1 TO NUMBER_OF_BPPS PARALLELLY DO BEGIN
   (** Interface with Predecessors and Successors **)
   IF (GPFS[i].Num_Clauses < Memory_Capacity) THEN BEGIN
      IF (GPFS[i].Mail = Clear) THEN BEGIN
         Request_Predecessors_for_Additional_Clauses;
         GPFS[i].Mail = Send_Me;
         END
      ELSE IF (GPFS[i].Mail = Connect) THEN BEGIN
         (* Receive a product at every cycle *)
         Received_Product :=
                     Receive_a_Product_from_Predecessors(i);
         IF (Received_Product <> Empty_Product) THEN BEGIN
            Allocate_Memory_and_Store_New_Products;
            Product_Count := Product_Count + 1;
            END (* if not Empty_Product *)
         ELSE  Clause_Count := Clause_Count + 1;
         END; (* in Connect state *)
      END
```

```
    ELSE BEGIN
       Setup_and_Deliver_an_Excessive_Clause_to_Successor;
       Clause_Count := Clause_Count + 1;
       END;

    (** Control CPG - Select SAPA - Coordinate Processes **)
    IF (GPFS[i].CPG.Status = DONE) THEN BEGIN
       IF (GPFS[i].Num_Clauses >= 2) THEN BEGIN
           Setup_a_Pair_of_Clauses_for_CPG_Operations;
           Clause_Count := Clause_Count + 1;
           GPFS[i].CPG.Status := BUSY;
           Select_Alternate_SAPA_in_the_SPRU;
           END;
       END
    ELSE BEGIN
       Collect_Partial_Optimal_Products_from_the_SPRU;
       Product_Count := Product_Count + 1;
       Clause_Count := Clause_Count + 1;
       END;
    END;
END. (** The Control Unit Algorithm **)
```

The CPG Algorithm

```
BEGIN
IF (GPFS[i].CPG.Status = BUSY) THEN BEGIN
   IF NOT (End_of_the_First_Clause) THEN BEGIN
      Product1 := Get_a_Product_from_1st_Clause;  (* in AR *)
      IF NOT (End_of_the_Second_Clause) THEN BEGIN
         Product2 := Get_a_Product_from_2nd_Clause;(* in R *)
         Boolean_Product := Product1  BITWISE_AND  Product2;
         END
      ELSE BEGIN
         Reset_Pointer_to_Start_of_Second_Clause;
         Advance_Pointer_to_Next_Product_in_First_Clause;
         END;
      END

   ELSE BEGIN (* Complete the multiplication *)
      IF (Number_of_Boolean_Products_Generated = 0) THEN BEGIN
         Termination_Status = ABNORMAL;
         GPFS_Finish = TRUE;
         END
      ELSE  GPFS[i].CPG.Status := DONE;
      END;
   END;
END. (** The CPG Algorithm **)
```

The SPRU Algorithm

```
BEGIN
(** Alter SAPA functions when a new clause is encountered **)
IF ((Alternate_SAPA = 1) AND
   (Left_SAPA = Current_Input_Device) AND
   (Right_SAPA.Status = Empty)) THEN BEGIN
   Reset_the_Right_SAPA;
   Right_SAPA := Current_Input_Device;
   Left_SAPA := Current_Output_Device;
   END
ELSE IF ((Alternate_SAPA = 0) AND
   (Right_SAPA = Current_Input_Device) AND
   (Left_SAPA.Status = Empty)) THEN BEGIN
   Reset_the_Left_SAPA;
   Left_SAPA := Current_Input_Device;
   Right_SAPA := Current_Output_Device;
   END;

(** Process received clauses - Get/Return products to PMU **)
FOR (Left and Right SAPA in SPRU) PARALLELLY DO BEGIN
   Compute_Cost_and_Check_Contradiction (Boolean_Product);
   IF (Boolean_Product[Tag_Bit] <> 0) THEN BEGIN
      Send_Boolean_Products_to_SPRU (Boolean_Product);
      Update_Number_of_Generated_Boolean_Products;
      END;
   ELSE  Discard_a_Product (Boolean_Product);
```

```
   IF (Alternate_SAPA = 0) THEN BEGIN
      Left_SAPA_Get_a_Boolean_Product_from_CPG;
      Left_SAPA_Process_Input_Phase_Operations;
      Right_SAPA_Process_Output_Phase_Operations;
      Right_SAPA_Return_a_Sorted_Product_to_PMU;
      END
   ELSE BEGIN (* Right SAPA is the input device *)
      Right_SAPA_Get_a_Boolean_Product_from_CPG;
      Right_SAPA_Process_Input_Phase_Operations;
      Left_SAPA_Process_Output_Phase_Operations;
      Left_SAPA_Return_a_Sorted_Product_to_PMU;
      END;
   END;
END. (** The SPRU Algorithm **)
```

The SAPA Algorithm

```
BEGIN
(** Initialization Phase: System RESET. N/2 PEs. One cycle **)
FOR i:=1 TO N/2 PARALLELLY DO BEGIN
    P[i] := MAX; (* Initialize registers to Empty Products *)
    Q[i] := MAX; (* containing a "1" in every bit position *)
    END;

(** INPUT Phase: unordered sequence of N items. N cycles. **)
FOR i:=1 TO N DO BEGIN
    FOR j:=1 TO N/2 PARALLELLY DO BEGIN
        Q[0] := Input_Datum;
```

```
        Q[j] := Q[j-1];
        IF (Q[j] < P[j]) THEN BEGIN (* Compare Costs *)
            P[j] := Q[j];  (* Swap register contents *)
            Q[j] := P[j];  (* Smaller item in P *)
            Check_Out_Dominating_and_Redundant_Products
                    (P[j], Q[j]);  (* by the PDD unit *)
            END;(* Find PDD description in Appendix A *)
        END;
     END;

(** OUTPUT Phase: ordered sequence of N items. N cycles. **)
FOR i:=1 TO N DO BEGIN
    FOR j:=1 TO N/2 PARALLELLY DO BEGIN
        IF (Q[j] < P[j]) THEN BEGIN (* Compare Costs *)
            P[j] := Q[j];  (* Swap register contents *)
            Q[j] := P[j];  (Smaller item in P *)
            Check_Out_Dominating_and_Redundant_Products
                     (P[j], Q[j]); (* by the PDD unit *)
            END;(* Find PDD description in Appendix A *)
         P[j] := P[j+1];
         P[0] := P[1]; (* Output_Datum *)
         END;
     END;
END; (** The SAPA Algorithm **)
```

VERIFICATION

The correctness and efficiency of the applied algorithms performed on the GPFS can be verified by software simulation. Figure 8 represents a simple example of the GPFS operations. Since the given GPF contains only four clauses, the Host fetches them into the leaf node BPP_1 and BPP_2. These BPPs simultaneously expand the given clauses, and send the sequences of Boolean products to the associated SPRUs. These SPRUs individually sort these sequences of Boolean products based on the product costs, and reduce the size of the expressions by eliminating all dominating and contradictory products. As illustrated, the SPRU in BPP_1 detects and removes ABD'F and ACDF, as they dominate AF, and BB'CD'E, as it is a contradictory product. On the other hand, the SPRU in BPP_2 detects no such products in its expression. As the remaining products in each sequence are sorted, each BPP selects and transfers the three least cost products to BPP_0. The root node processor multiplies out the two given clauses and repeats the reduction processes. Again, the three most optimal products selected are to be returned back to the Host: AFH'P and AFN'P (of cost 4), and BD'FH'PN (of cost 6) in this example.

The GPFS Simulator has been developed to verify the correctness and the efficiency of the algorithms executed on the GPFS architecture. Also, this simulation program can be

used as a tool for making design trade-off decisions between the system performance and the optimality of the design. Since the GPFS is a data dependent architecture, typical optimization criteria shall include the execution time, the number of DFT levels, the size of the linear array of PEs in each SAPA, the pipelining period, the block pipelining period, the handshaking methods for intercommunication between the BPPs at successive levels of the tree, the I/O channels, and the processor utilization, which depends on all of the above optimality factors [Kung, 1988]. The above design optimization criteria will be discussed further in the next chapter together with a detailed discussion regarding the GPFS program. A set of simulation test cases has been developed to examine different aspects of the GPFS system behavior in response to various combinations of the system configuration parameters (e.g., the SAPA size, the number of optimal products selected from each clause processed by the SAPA, the size of the local memory storage in each BPP, and others).

REMARKS

According to the definition given by Kung [1988], the overall GPFS system is a wavefront array processor which contains the following four features: "self-timed, data-driven computation; regularity, modularity and local interconnection; programmability in wavefront language or data flow graph; and pipelinability with linear-rate speed-up". The most important

and distinctive characteristic of this architecture is the data-driven operation of each BPP. The system adopts a two-way asynchronous communication control scheme to properly

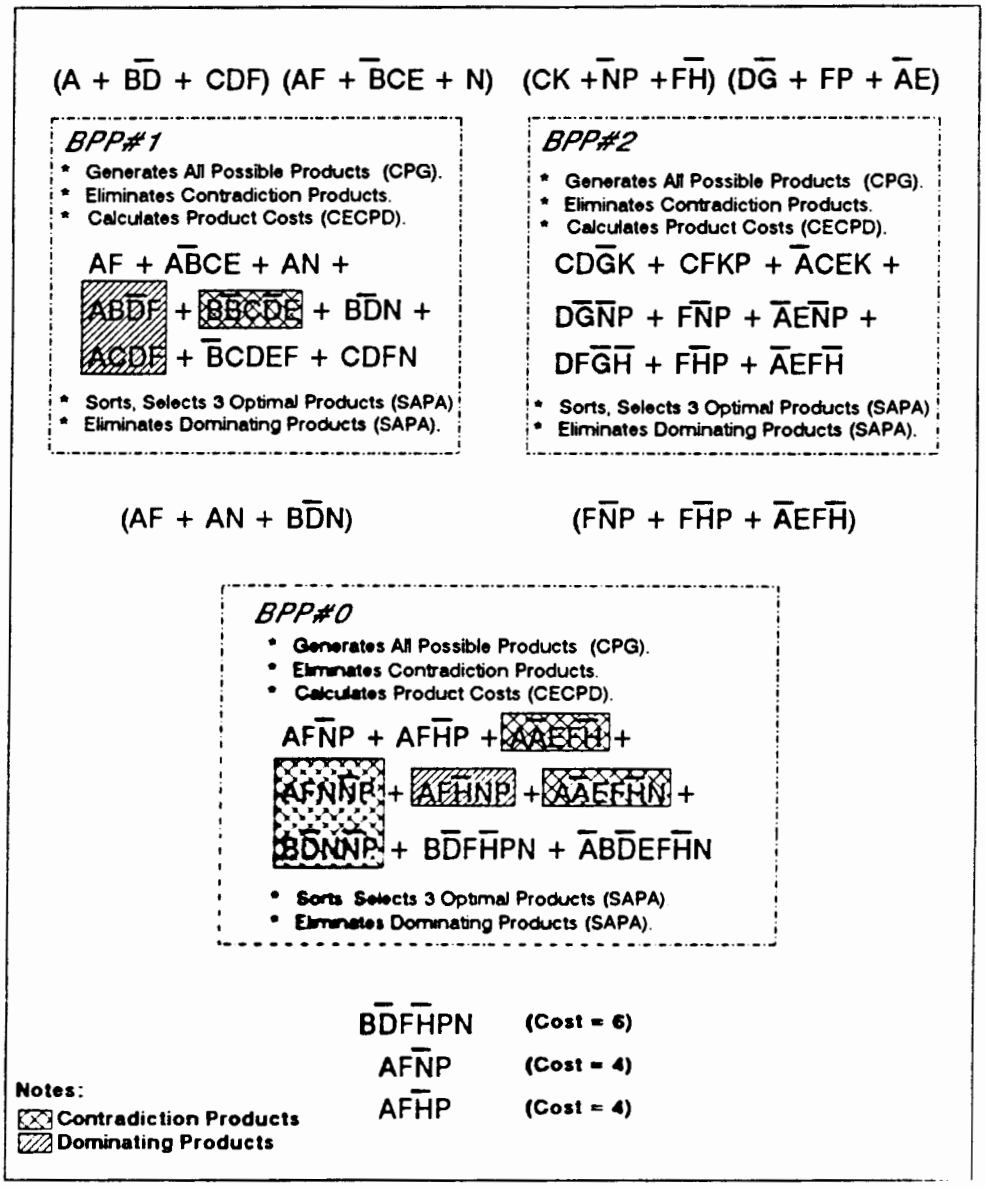


Figure 8. The GPFS operations - an example.

synchronize the operations, to correctly control the sequences of data transfers, and to avoid possible bus contention problems. These simple yet efficient handshaking protocols only require the senders' and receivers' acknowledged signals at the start, and at the end, of a series of data transmission (i.e., a GPF clause transfer).

The SAPA, on the other hand, is a systolic array that contains these four major features: "synchrony; modularity and regularity; spacial locality and temporal locality; and linear rate pipelinability" [Kung,1988]. This architecture, although requiring an accurate global clock, provides other preferable characteristics such as its simple, regular circuit, high degrees of computing concurrency, short local interconnections, and balanced I/O bandwidth with the master processor. All these features make a systolic array processor suitable for a special purpose computing accelerator such as the SAPA in this application.

As a wavefront array processor, the GPFS possesses most of the advantages of the systolic array processor. Moreover, the fact that the GPFS does not need a global system clock frees the designers from dealing with critical timing problems in VLSI array system design. Finally, the programmability feature make this architecture fully scalable; the computational wavefronts and the asynchronous timing capability significantly simplify the task of programming for this parallel computing systems.

CHAPTER IV

THE GPFS SIMULATION AND EVALUATION

In order to obtain a set of simulation test cases which can be used to accurately evaluate the effectiveness of the GPFS architecture, let us consider the most crucial issues which directly affect the performance and the optimality of the design. Discussions regarding the GPFS simulation program, the test cases and results, and the evaluation are presented in the subsequent sections.

As a wavefront array processor, the efficiency of the GPFS processing can be expressed in terms of Concurrency and Communication [Kung, 1988]. In the GPFS, the concurrency is achieved by

(1) Programming the Host to decompose the given GPF into subproblems, and fetch the subproblems to the leaf node processors in an interleaved manner.

(2) Maximizing the parallelism and pipelining of the entire system: the BPPs at every level of the DFT should be kept busy at all time, and the SAPAs should have their pipelines rhythmically filled up with meaningful data at every execution cycle.

(3) Optimizing the intercommunications with simple and effective handshaking protocols: the protocols should

impose minimal transition delays to consequently allow the communication partners to acknowledge the sending messages in the shortest possible length of time.

Obviously, the concurrency and the communication factors mutually compensate for each other: effective communications provide the BPPs with sufficient data to continuously perform their parallel, pipeline processes, and to effectively transfer the excessive data from local memory storage to processors on successive levels. On the other hand, the higher degree of concurrency increases the computation rate, and subsequently optimizes the I/O bandwidths of the system. In a closed loop operating mode (i.e., Host - leaf level BPPs - intermediate level BPPs - root level BPP - Host), with the flow of a larger number of data streams due to higher degree of parallelism, data in the system will be distributed more uniformly to all processors in the DFT.

It is usually difficult to make decisions for trade-offs between system performance and the optimality of the design. For many practical circumstances, the decisions are made based on the specific target applications. Typical optimization factors for a data-driven architecture, such as the GPFS, shall include the following basic criteria:

(1) The execution time: the number of execution steps that the GPFS requires to completely resolve a GPF problem.

(2) The pipelining time: the length of time required to obtain the first pair of clauses of Boolean products to

the root node BPP (DFT pipelining time), and the time to fill up the SAPA in each BPP (SAPA pipelining time). The DFT pipelining time is dependent on the number of DFT levels and the data nature of the given GPF. The SAPA pipelining time is directly proportional to the number of PEs in its linear systolic array.

(3) The number of DFT levels: The greater the number of levels, the larger the number of BPPs. This parameter directly affects the implementation cost, and increases the complexity of balancing the processing loads on the BPPs at different tree levels. However, for large GPF problems, it is appropriate to increase this parameter to speed-up the process with additional processors.

(4) The size of SAPA: Besides the direct impact on implementation cost and computation speed, the number of PEs in each SAPA also affects the quality (cost) of the solutions. Since the main functions of the SAPA is to rearrange and select the most optimal products from the sequences of Boolean products received from the CPG, the greater the number of PEs in each SAPA, the greater the number of Boolean products (of the same clauses) are compared with each other. Consequently, the better optimal products can be found.

(5) The I/O channels: The number of interconnections between the Host and the GPFS. In this application, the data buses between the Host and the connected processors are

unidirectional. Being connected to the root node and leaf node BPPs through its I/O ports, the Host transfers the clauses of the given GPF to the leaf node processors in a time-shared, interleaved fashion, and at the same time, accepts the returning clauses from the root node BPP. Since the processes performed on the CPG and the SAPA usually require much longer time than the time to transfer a clause from one processor to another, the GPFS can be attached directly to a Host with conventional I/O ports. However, to receive the 256-bit Boolean products returned from the root node BPP, an I/O Interface Processor (or a buffer) could be used.

(6) The processor utilization: This factor determines the efficiency of the system and the effectiveness of employing a certain number of processors in the system realization. The processor utilization factor is dependent on all optimality factors described above, and can be observed as a direct measurement of the balance in processing loads among the BPPs [Kung, 1988].

THE GPFS SIMULATOR

The GPFS Simulation program is divided into four modules which, individually, contain the procedures corresponding to the GPFS major tasks described in the previous chapter: HOST, BPP, PMU, and SAPA. The HOST module includes the Program Sequencer (MAIN) and procedures to

- Initialize the parameters that are relevant to the Host-GPFS communication process (Init-Host).
- Read input data from an input file, and encode them into 256-bit patterns (Get-Input-Data, Product-Encoder).
- Interface to the root node and leaf node BPPs. This part includes procedures for updating the Host status, setting up pointers to transfer Boolean products to the leaf node BPPs, and receiving products from the root node BPP (HOST-Update-Mail-Status, HOST-Setup-Transfer-Header, HOST-Fetch-Leaf-Node-BPPs, HOST-Leaf-BPPs-Interface, and HOST-Receive-Root-BPP-Products).
- Generate the operation reports and handle miscellaneous tasks (Product-Decoder, GPFS-Report, GPFS-Final-Report, and other utility subroutines).

The BPP module includes procedures necessary to create the dynamic data structure for

- Modeling the BPP, (Create-BPP-Node, Create GPFS).
- Simulating the functions of the Control Unit in the BPP (BPP-Operations, BPP-Interface-with-Predecessors, BPP-Support-Successors, and Accept-and-Link-SAPA-Results).

For the PMU processes, the PMU module consists of:

- Procedure PMU-Operation, the main executive which invokes other subroutines to perform the Cartesian Product Generating process, compute the costs of the generated products, detect contradictory products, handle data transfers between the local memory storage and the CPG,

and control the CPG - SPRU communication.

- Procedures CPG-Operations, Generate-a-Boolean-Product, Cost-Contradiction-Checker, Transfer-Clauses-to-CPG, and Transfer-a-Product-Node respectively perform their namely descriptive functions.

The SAPA module contains the procedures necessary to perform the operations of the Sum of Product Reduction Unit, which includes the tasks of assigning the I/O function to the SAPAs, sorting/absorbing the given product sequences, and select the least cost product from the sorted SOP expressions. Being the main procedure in this module, the SPRU-Operation invokes the SAPA-Input-New-Items, SAPA-Output-Sorted-Items, SAPA-Compare-and-Swap routines to sort the products into descending order. The procedures SAPA-Indicate-Dominating-Product, and SAPA-Reset are executed to simplify the returning sequences, and to initialize the SAPA for processing another sequence.

To evaluate the GPFS performance, a program has been written to generate the Generalized Propositional Formula of various sizes for testing purposes (program GENGPF). This program provides the capabilities to select the number of variables in the GPF, the number of clauses in the GPF, the maximum number of products per clause, the maximum number of literals per product, and the percentage of the number of negated versus non-negated literals in the given expression.

Appendices B, and C respectively contain the source listings of the GPFS program, the random GPF test case

generator, GENGPF, and a sample script file including a test case and the corresponding test results.

TEST CASES AND RESULTS

A number of randomly generated GPF expressions have been tested on different GPFS configurations. The test results are summarized in the following tables according to the design criteria described in the first part of this chapter. The GPFS simulation is segregated into 10 tests, which separately exhibit different architectural aspects of the system: the SAPA Size Analysis, the DFT Structure Analysis, the Problem Characteristics Analysis, the PMU Local Memory Size Analysis, the Host-GPFS Communication Analysis, and the Processor Utilization Analysis.

In the DFT Structure Analysis, a set of five test trials is executed on four GPFS configurations differentiated by the number of DFT levels. In Test 1, the problem size is fixed at 250 clauses. The SAPA size is fixed at 64. The number of DFT levels is varied from 2 to 5. Table II contains the test results where each entry is a rounded average value of the results of five test trials.

Tables III and IV contains the average results of the SAPA Size Analysis. In Tests 2, a set of five problem instances is executed on a GPFS configured with 2 DFT levels. The problem size is fixed at 50. In Test 3, another set of 5 problems is tested on a 3-DFT-level GPFS. Each GPF problem

in the set contains 100 clauses. In both cases, the SAPA size is varied between 16 and 100. This analysis is useful to determine the optimum number of PEs for implementing the SAPA. Reducing the SAPA size shall result in proportionally accelerating the system operation and directly decreasing the implementation cost.

Tables V, VI, and VII contain the results of the Problem Characteristics Analysis: in Test 4, the problem size is varied between 50 and 1000 clauses while the number of DFT levels is fixed at 2, the SAPA size is set at 100, and the percentage of negated literals is set to zero. In Tests 5 and 6, the percentage of the negated literals in the given expressions increases from 0% to 10%, and from 90% to 100%. The number of the DFT levels is kept constant at 2 and the SAPA size is fixed at 36, whereas the numbers of clauses in the GPF are set to 50 and 100, respectively.

The results of the PMU Local Memory Size Analysis is presented in Table VIII: each test problem includes 1000 clauses; where each clause contains at most 4 Boolean products, and each of these Boolean product contains at most 4 positive literals. In this combination, the memory block required to accommodate each clause of the given GPF problem is approximately 1 KiloBytes (i.e., 256 bytes/clause * 4 clauses/problem = 1024 bytes = 1 KB). The number of DFT levels and the SAPA size are fixed at 2 and 25, respectively. The size of local memory storage is varied between 2 KB and

16 KB, which respectively corresponds to 2 to 16 clauses accommodatable in the PMU local memory at any instant of time. The purpose of Test 7 is to optimize the size of memory storage which will effectually accelerate the system operations with minimum sacrifice to the system performance due to the processor load balance.

In the Host-GPFS Communication Analysis, a set of test trials are derived to inspect the system performance due to the changes in number of BPPs in the GPFS and the number of I/O channels established between the Host and the GPFS at every execution cycle. The problem size is fixed at 1000 clauses, the SAPA size is settled at 16, while the number of DFT levels is varied from 2 to 5, which correspond to 2, 4, 8, and 16 BPPs in the leaf-level of the DFT, separately. The number of BPPs can be fetched by the Host at every GPFS execution cycle is varied from 25%, 50%, to 100% of the total of BPPs existing at the leaf level. The same set of problem instances is repeated with different number of DFT levels in the GPFS. The results of the tests (8, 9, and 10) are presented in Tables IX, X, and XI. The ratio of the actual execution time required (EXETIME, in seconds) versus the total execution cycles (EXECYCL) provides an additional scale to evaluate the efficiency of the communication between the Host and the GPFS. The larger ratio signifies that more time is taken to transfer clauses from one DFT level to the next.

The Processor Utilization Analysis reveals the efficiency

of the intercommunication between the BPPs at adjacent levels of the DFT, and the distribution of processing loads among the BPPs at different DFT levels in the system. This analysis provides a useful measurement to achieve a cost-effective design and to maximize the system throughput by balancing the processing loads and keeping all the BPPs in the GPFS in active state continuously (i.e., minimizing the number of BPPs idle cycles). Since the processor utilization factor is not only dependent on the nature of the problem, but also on the SAPA size, the size of the PMU local memory storage, the number of DFT levels, and the number of communication channels established between the Host and the leaf node BPPs, the results of the test cases shown in Tables II to XI are altogether taken into consideration. Moreover, the percentage of the number of active cycles versus the total number of execution cycles of the BPPs at each DFT level can be used to determine the efficiency of utilizing the system hardware (i.e., UTIL = [1 -IDLE/EXECYCL] x 100%). It would be desirable to keep this parameter as large as possible.

Except for Test 7, where the sizes of the PMU local memory are explicitly specified, in all other tests, the capacity of PMU local memory storage is set equivalent to the size of 4 clauses of the given problem.

Each table of results can be separated into three parts: the first part contains the GPF problem specification, which includes the number of literals (NLIT), the number of clauses

(NCLAU) in the given GPF, the number of possible products per clause (MCS: Max Clause Size), the number of possible literals per product (MPS: Max Product Size), and the percentage of negated literals in the given expression (NEG). The second part contains the GPFS structural specification and the status of the BPPs at time the GPFS execution is terminated. This part contains the SAPA size (SAPA), the number of levels in the tree (DFTS), the number of BPPs in the system (BPPS), the number of clauses transferred to the next DFT level (XFER), the number of clauses received from the predecessors at the previous tree level (RCVD), the number of idle cycles (IDLE), and the percentage of utilization (UTIL) of the BPPs at each level of the DFT during the entire course of execution. The last part of the table contains the number of clauses that the Host received from the root node BPP (FEEDBK), the minimum cost of the results (COST), the total execution cycles (EXECYCL), and the execution time (EXETIME, in seconds) required to run the test on an IBM PC-AT operating at 10 MHz. Since the final solutions are expressed in the form of SOP expressions, only the costs of the most optimal products are shown in the tables; a cost of zero indicates that no solution for the given GPF exists.

TABLE II

TEST 1 - THE DFT STRUCTURE ANALYSIS

GPFS SYSTEM STATUS	NUMBER OF DFT LEVELS			
	2	3	4	5
GPF				
NLIT	124	124	124	124
NCLAU	250	250	250	250
MCS	8	8	8	8
MPS	8	8	8	8
NEG	0	0	0	0
BPPS	3	7	15	31
SAPA	64	64	64	64
DFT 0				
RCVD	585	222	77	41
XFER	426	145	34	1
IDLE	33	66	101	124
UTIL	99.3	97.5	96.7	93.5
DFT 1				
RCVD	422	151	62	35
XFER	253	91	28	10
IDLE	571	660	827	711
UTIL	87.5	75.6	61.9	63.3
DFT 2				
RCVD		136	51	31
XFER		60	22	11
IDLE		1093	1260	1142
UTIL		59.3	41.7	40.5
DFT 3				
RCVD			52	26
XFER			18	10
IDLE			1557	1475
UTIL			27.9	22.9
DFT 4				
RCVD				21
XFER				9
IDLE				1723
UTIL				9.9
FEEDBK	426	145	34	1
COST	119	118	120	120
EXECYCL	4544	2662	2157	1911
EXETIME	221.1	185.0	237.1	253.8

TABLE III

TEST 2 - THE SAPA SIZE ANALYSIS

GPFS SYSTEM STATUS	S A P A S I Z E S						
	16	25	36	49	64	81	100
GPF							
NLIT	124	124	124	124	124	124	124
NCLAU	50	50	50	50	50	50	50
MCS	8	8	8	8	8	8	8
MPS	8	8	8	8	8	8	8
NEG	0	0	0	0	0	0	0
DFTS	2	2	2	2	2	2	2
BPPS	3	3	3	3	3	3	3
DFT 0							
RCVD	66	72	72	81	90	93	96
XFER	26	31	33	44	53	54	58
IDLE	32	32	32	34	33	32	32
UTIL	92.9	95.4	96.3	96.3	97.5	97.9	98.4
DFT 1							
RCVD	52	55	56	62	67	67	68
XFER	23	26	26	31	36	37	38
IDLE	182	319	446	499	756	833	1162
UTIL	60.5	55.2	49.4	50.5	45.0	46.2	41.4
FEEDBK	26	31	33	44	53	54	58
COST	69	69	68	67	67	67	67
EXECYCL	460	713	881	1009	1370	1544	1988
EXETIME	7.7	14.7	20.7	28.5	46.1	61.5	63.4

TABLE IV

TEST 3 - THE SAPA SIZE ANALYSIS

GPFS SYSTEM STATUS	S A P A S I Z E S						
	16	25	36	49	64	81	100
GPF							
NLIT	120	120	120	120	120	120	120
NCLAU	100	100	100	100	100	100	100
MCS	16	16	16	16	16	16	16
MPS	8	8	8	8	8	8	8
NEG	0	0	0	0	0	0	0
DFTS	3	3	3	3	3	3	3
BPPS	7	7	7	7	7	7	7
DFT 0							
RCVD	58	64	64	67	69	72	72
XFER	15	22	25	25	31	33	34
IDLE	53	54	55	57	64	64	64
UTIL	89.9	94.5	94.4	95.4	96.1	96.9	97.4
DFT 1							
RCVD	47	50	51	52	53	56	56
XFER	18	21	22	23	25	26	26
IDLE	237	377	434	614	752	1069	1215
UTIL	55.7	55.7	57.1	51.3	54.3	48.1	51.0
DFT 2							
RCVD	40	42	43	43	45	45	46
XFER	16	17	18	18	19	20	20
IDLE	303	540	647	865	1133	1518	1803
UTIL	43.4	36.2	35.7	31.3	30.9	26.2	27.3
FEEDBK	15	22	25	25	31	33	34
COST	112	112	111	111	111	110	110
EXECYCL	535	845	1005	1257	1635	2058	2479
EXETIME	15.6	29.6	42.2	58.4	97.0	132.5	201.7

TABLE V

TEST 4 - THE PROBLEM CHARACTERISTICS ANALYSIS

GPFS SYSTEM STATUS	P R O B L E M S I Z E S						
	50	100	150	200	500	750	1000
GPF							
NLIT	124	124	124	124	124	124	124
MCS	8	8	8	8	8	8	8
MPS	8	8	8	8	8	8	8
NEG	0	0	0	0	0	0	0
DFTS	2	2	2	2	2	2	2
BPPS	3	3	3	3	3	3	3
SAPA	100	100	100	100	100	100	100
DFT 0							
RCVD	96	253	414	570	1467	2186	2910
XFER	58	183	309	438	1130	1668	2218
IDLE	32	32	32	32	33	33	37
UTIL	98.3	99.0	99.2	99.4	99.7	99.8	99.8
DFT 1							
RCVD	68	173	278	386	980	1454	1935
XFER	38	109	181	252	649	963	1282
IDLE	1162	1359	1063	1157	652	707	649
UTIL	41.4	58.6	75.0	79.0	94.1	95.4	96.7
FEEDBK	58	183	309	438	1130	1668	2218
COST	68	96	108	115	122	123	123
EXECYCL	1988	3278	4229	5443	10923	15483	20120
EXETIME	93.4	179.0	268.1	357.8	776.8	1102.3	1436.4

TABLE VI

TEST 5 - THE PROBLEM CHARACTERISTICS ANALYSIS

GPFS SYSTEM STATUS	PERCENTAGE OF NEGATED LITERALS									
	1	3	5	7	10	90	92	95	97	100
GPF										
NLIT	120	120	120	120	120	120	120	120	120	120
NCLAU	50	50	50	50	50	50	50	50	50	50
MCS	5	5	5	5	5	5	5	5	5	5
MPS	5	5	5	5	5	5	5	5	5	5
DFTS	2	2	2	2	2	2	2	2	2	2
BPPS	3	3	3	3	3	3	3	3	3	3
SAPA	36	36	36	36	36	36	36	36	36	36
DFT 0										
RCVD	65	65	66	64	65	65	57	66	65	65
XFER	28	28	25	28	25	25	24	25	28	28
IDLE	26	26	26	26	26	26	24	26	26	26
UTIL	96.4	96.4	96.4	96.3	96.3	96.3	96.5	96.4	96.4	96.4
DFT 1										
RCVD	54	54	52	54	52	52	53	52	54	54
XFER	23	23	23	23	23	23	20	23	23	23
IDLE	427	427	432	416	415	415	383	432	427	427
UTIL	40.5	40.5	59.8	41.1	40.8	40.8	43.7	39.8	40.5	40.5
FEEDBK	28	28	25	28	25	25	24	25	28	28
COST	57	58	58	0	0	0	56	58	58	57
EXECYCL	718	718	718	707	701	701	680	718	718	718
EXETIME	15.2	15.2	13.8	14.4	13.5	13.5	14.7	13.8	15.2	15.2

TABLE VII

TEST 6 - THE PROBLEM CHARACTERISTICS ANALYSIS

GPFS SYSTEM STATUS	PERCENTAGE OF NEGATED LITERALS									
	0	1	2	3	5	95	97	98	99	100
GPF										
NLIT	120	120	120	120	120	120	120	120	120	120
NCLAU	100	100	100	100	100	100	100	100	100	100
MCS	5	5	5	5	5	5	5	5	5	5
MPS	5	5	5	5	5	5	5	5	5	5
DFTS	2	2	2	2	2	2	2	2	2	2
BPPS	3	3	3	3	3	3	3	3	3	3
SAPA	36	36	36	36	36	36	36	36	36	36
DFT 0										
RCVD	150	150	149	146	159	159	146	149	150	150
XFER	81	81	81	81	78	78	81	81	81	81
IDLE	33	33	33	33	33	33	33	33	33	33
UTIL	97.1	97.1	97.0	98.2	96.6	96.6	96.2	97.0	97.1	97.1
DFT 1										
RCVD	123	123	123	123	118	118	123	123	123	123
XFER	58	58	58	58	60	60	58	58	58	58
IDLE	479	480	417	210	328	328	210	417	480	479
UTIL	58.3	58.2	51.6	76.1	66.5	66.5	76.1	61.6	58.2	58.3
FEEDBK	81	81	81	81	78	78	81	81	81	81
COST	84	84	0	0	0	0	0	0	84	84
EXECYCL	1148	1149	1086	879	979	979	879	1086	1149	1148
EXETIME	28.6	28.3	27.7	24.1	24.8	24.8	24.1	27.7	28.3	28.6

TABLE VIII

TEST 7 - THE PMU LOCAL MEMORY SIZE ANALYSIS

GPFS SYSTEM STATUS	MEMORY CAPACITY [in KILOBYTES]					
	2	3	4	6	12	16
GPF						
NLIT	120	120	120	120	120	120
NCLAU	1000	1000	1000	1000	1000	1000
MCS	4	4	4	4	4	4
MPS	4	4	4	4	4	4
NEG	0	0	0	0	0	0
DFTS	3	3	3	3	3	3
BPPS	7	7	7	7	7	7
SAPA	25	25	25	25	25	25
DFT 0						
RCVD	379	531	569	577	582	584
XFER	192	294	324	322	309	306
IDLE	149	62	50	53	46	46
UTIL	93.9	97.4	98.0	97.8	98.2	98.3
DFT 1						
RCVD	354	436	466	466	465	471
XFER	142	206	223	225	222	222
IDLE	136	137	201	227	235	349
UTIL	94.5	94.3	91.9	91.0	87.5	87.0
DFT 2						
RCVD	471	486	489	488	482	479
XFER	124	160	172	172	171	173
IDLE	179	201	294	349	571	605
UTIL	92.8	91.7	88.2	86.1	78.7	77.4
FEEDBK	192	294	324	322	309	306
COST	119	119	119	119	119	119
EXECYCL	2467	2408	2480	2485	2652	2669
EXETIME	141.4	142.9	143.6	139.4	140.0	142.6

TABLE IX

TEST 8 - THE HOST-GPFS COMMUNICATION ANALYSIS

GPFS SYSTEM STATUS	NUMBER OF HOST CHANNELS / NUMBER OF LEAF-BPPS					
	1/4	2/8	4/16	1/4	2/8	4/16
GPF						
NCLAU	750	750	750	1000	1000	1000
NLIT	120	120	120	120	120	120
MCS	12	12	12	4	4	4
MPS	5	5	5	4	4	4
NEG	0	0	0	0	0	0
DFTS	3	4	5	3	4	5
BPPS	7	15	31	7	15	31
SAPA	16	16	16	16	16	16
DFT 0						
RCVD	63	13	17	68	18	19
XFER	2	1	1	3	1	1
IDLE	2163	1344	746	2425	1507	829
UTIL	20.6	5.4	10.9	21.9	8.7	13.2
DFT 1						
RCVD	150	34	13	168	37	13
XFER	16	3	4	18	4	5
IDLE	1164	1068	743	1426	1179	840
UTIL	57.3	24.8	11.3	54.1	28.5	12.0
DFT 2						
RCVD	334	76	204	455	86	20
XFER	41	9	4	46	10	4
IDLE	591	572	610	664	695	720
UTIL	88.3	59.8	27.3	78.6	57.9	24.6
DFT 3						
RCVD		165	38		225	42
XFER		21	5		24	5
IDLE		356	427		430	500
UTIL		74.9	49.1		73.9	47.6
DFT 4						
RCVD			82			112
XFER			10			11
IDLE			314			350
UTIL			62.5			63.3
FEEDBK	2	1	1	3	1	1
COST	119	119	119	119	119	119
EXECYCL	2723	1419	837	3105	1647	953
EXETIME	69.5	73.6	76.1	75.7	81.7	83.1

TABLE X

TEST 9 - THE HOST-GPFS COMMUNICATION ANALYSIS

GPFS SYSTEM STATUS	NUMBER OF HOST CHANNELS / NUMBER OF LEAF-BPPS							
	1/2	2/4	4/8	8/16	1/2	2/4	4/8	8/16
GPF								
NCLAU	750	750	750	750	1000	1000	1000	1000
NLIT	120	120	120	120	120	120	120	120
MCS	12	12	12	12	4	4	4	4
MPS	5	5	5	5	4	4	4	4
NEG	0	0	0	0	0	0	0	0
DFTS	2	3	4	5	2	3	4	5
BPPS	3	7	15	31	3	7	15	31
SAPA	16	16	16	16	16	16	16	16
DFT 0								
RCVD	614	261	147	77	742	294	161	75
XFER	203	99	58	21	233	104	59	22
IDLE	794	361	183	173	954	404	202	191
UTIL	77.7	78.9	79.7	71.2	76.5	78.6	80.7	71.3
DFT 1								
RCVD	748	297	137	70	988	343	152	70
XFER	204	90	51	24	244	99	55	24
IDLE	162	224	174	192	171	236	204	202
UTIL	95.5	86.9	80.8	68.2	95.8	87.6	80.5	69.7
DFT 2								
RCVD		327	139	62		440	163	69
XFER		96	46	23		110	51	23
IDLE		181	184	218		174	251	253
UTIL		89.4	79.7	64.0		90.9	76.2	62.0
DFT 3								
RCVD			155	63			210	77
XFER			46	21			53	23
IDLE			179	265			227	267
UTIL			80.3	56.2			78.4	60.0
DFT 4								
RCVD				74				102
XFER				20				24
IDLE				257				273
UTIL				57.5				59.0
FEEDBK	203	99	58	21	233	104	59	22
COST	119	119	119	119	119	119	119	119
EXECYCL	3563	1711	906	602	4056	1894	1049	665
EXETIME	59.0	67.6	72.3	74.3	64.8	73.4	79.0	81.3

TABLE XI

TEST 10 - THE HOST-GPFS COMMUNICATION ANALYSIS

GPFS SYSTEM STATUS	NUMBER OF HOST CHANNELS / NUMBER OF LEAF-BPPS							
	2/2	4/4	8/8	16/16	2/2	4/4	8/8	16/16
GPF								
NCLAU	750	750	750	750	1000	1000	1000	1000
NLIT	120	120	120	120	120	120	120	120
MCS	12	12	12	12	4	4	4	4
MPS	5	5	5	5	4	4	4	4
NEG	0	0	0	0	0	0	0	0
DFTS	2	3	4	5	2	3	4	5
BPPS	3	7	15	31	3	7	15	31
SAPA	16	16	16	16	16	16	16	16
DFT 0								
RCVD	952	384	171	82	1189	478	201	93
XFER	442	182	74	20	547	228	91	28
IDLE	29	52	73	100	32	53	71	95
UTIL	99.0	96.5	90.9	81.7	99.1	96.9	92.2	84.2
DFT 1								
RCVD	843	342	151	70	1113	428	179	83
XFER	348	141	61	25	434	176	73	30
IDLE	90	113	100	108	84	139	103	115
UTIL	97.0	92.5	87.8	80.3	97.6	91.8	88.7	80.9
DFT 2								
RCVD		344	144	66		462	181	81
XFER		120	52	23		151	62	28
IDLE		172	152	151		193	170	182
UTIL		88.5	81.3	72.8		88.6	81.4	70.0
DFT 3								
RCVD			156	67			211	86
XFER			49	22			61	27
IDLE			195	220			219	238
UTIL			76.2	59.9			76.1	59.8
DFT 4								
RCVD				73				99
XFER				22				28
IDLE				266				289
UTIL				51.4				52.4
FEEDBK	442	182	74	20	547	228	91	28
COST	119	119	119	119	119	119	119	119
EXECYCL	3027	1495	814	547	3470	1698	913	606
EXETIME	63.4	69.1	71.7	74.2	70.8	76.7	79.0	81.0

EVALUATION

From Table II, the DFT Structure Analysis reveals that with a greater number of tree levels in the DFT, there are more processors in the systems; hence higher degrees of concurrency can be obtained and the process is accelerated: for GPF problems which contains 250 clauses, increasing the number of DFT levels from 2 to 3 reduces the total execution cycles from 4544 down to 2662 cycles. Although further increment to the number of DFT levels introduces extra acceleration to the process, the ratio of EXETIME / EXECYCL reveals that, for problems of this size, a GPFS with 2 DFT levels requires the least amount of time for data transfer between tree levels. Moreover, the UTIL factor indicates that the maximum processor utilization can be achieved with a DFT of size 2 for this test case. The "Area versus Time" rule shall be employed here to optimize the effectiveness of the trade-offs between hardware components and higher speed performance: as shown in Test 1, a two-level GPFS obtains a solution in 4544 cycles with its 3 processors, while a five-level GPFS obtains one in 1911 cycles with 31 processors. By taking the number of processors multiplied by the computation time, the two-level GPFS proves to be more cost effective than its counterpart in solving problems of this size.

From Tables III and IV, the SAPA Size Analysis indicates that with larger SAPA size, better solution can be obtained

with the trade-offs of execution time and implementation cost. Increasing the SAPA size decreases the average processor utilization (UTIL). The SPRU will take longer time to complete its processes with larger number of Boolean products in its longer queues, meanwhile the PMU will have to wait for the completion of a SAPA job prior to starting a new CPG process. Furthermore, with higher number of DFT levels and larger problem sizes, this issue becomes more apparent in cross referencing the results of Tests 2 and 3.

The results of the Problem Characteristics Analysis, shown in Tables V, VI, and VII illustrates further that, for problems of larger sizes, the GPFS can achieve better processor load balance and higher processor utilization efficiency: for smaller problems, the leaf node processors suffer from many inactive cycles (IDLE) in waiting for the BPPs at the successive levels to complete the process, while the Host has no more clauses to transfer to them. In large problems, more data transfers are performed: the root node BPP can complete a feedback loop to return excessive clauses from its local memory to the Host. The returned clauses (FEEDBK) will be made available for the leaf node BPPs in the next pass. As shown in Test 4, the processor utilization factor in a two-level GPFS is maximized in solving problems containing 500 clauses or more. Furthermore, since the GPFS is a data-driven system, the data nature of the problems should also be taken into consideration. The results of Tests

5 and 6 demonstrate that, for many cases, in GPF problems which contain 50 clauses and a number of negated literals between 8 and 92 percent of the total number of literals in the expression, there is usually NO solution. The same result is obtained for problems of size 100, with 2 to 98 percent of negated literals. It should be emphasized that, since the GPF is a general representation of the Satisfiability problem, which is a type of Decision problems, the capability of quickly indicating the nonexistence of a solution for a given problem is equally important as the ability to attain an optimal solution, if any.

The PMU Local Memory Size Analysis shows that, for large problems and small number of DFT levels (1000 clauses and 3 DFT levels, as in Test 7), the current of data flow (i.e., the number of clauses transferred) in the Host - GPFS - Host loop increases with respect to the expansion of the PMU local memory. As presented in Table VIII, the maximum data transfer rate is obtained with local memory capacity of 10 KB. Beyond that, the data flow is decreasing since the memory storage is sufficiently large to retain the clauses received from the predecessors, while perform the Cartesian Product Generating process and collect the results of the Sorting and Absorbing Processes. It can also be observed that, the optimum size for the PMU local memory storage in this test case is 3 KB: with a memory capacity equivalent to the size of 3 clauses, the processor utilization is maximized (averaged UTIL parameter

is 94.5) while the execution time is minimized (2408 cycles). These effects are obtained since the size constraint of the PMU local memory storage requires the BPPs at any level of the DFT in the system to transfer the fourth clauses to the successor nodes. As a result, all excessive clauses are returned back to the Host to be fetched to the leaf node BPPs. The process is accelerated because a higher degree of concurrency can be obtained with large number of nodes in the leaf-level of the DFT (the number of BPPs in the leaf level is equal to the total number of BPPs in all subsequent levels of the DFT plus one).

From the results of the Host-GPFS Communication Analysis, it can be observed that as more concurrent communication channels are established between the Host and the leaf node BPPs, higher current of data flows through the GPFS, which enhances the system throughput by increasing the utilization of the processors and the balance of processing loads among the BPPs at different system levels. As presented in Tables IX through XI, it is obvious that the larger communication bandwidth will accelerate the entire GPFS operations: increasing the Host capability to fetch 25% up to 100% of the number of BPPs in the leaf level, the total number of required execution cycles is reduced from 3105 down to 1698 for case of solving a 1000-clause problem on a three-level GPFS.

Although the number of DFT levels in the GPFS can be theoretically extended indefinitely, this expansibility is

restrained by the Host - GPFS communication bandwidth: the number of BPPs in the system must be limited proportionally to the number of BPPs that can be served by the Host at every execution cycle. Therefore, to determine the largest size of the DFT which a host can handle effectively, the Host - GPFS Communication Analysis should be performed. The results of the simulation will expose the correlation of the effectiveness of processor utilization and the growth of DFT levels: using an IBM PC-AT operating at 10 Mhz as the Host, with the number of DFT levels ranging from 2 to 5, the mean percentage of BPP active times are 86.2%, 85.7%, 78.9%, and 64.4%, as shown in Table X, for the case of the Host fetching 50% of the leaf node BPPs at every execution cycle. Now, if the Host can serve all the leaf node BPPs at every execution cycle, the mean BPP utilization factors are 98.3%, 92.4%, 84.6%, and 69.7%, as presented in Table XI.

From the results of the simulation, it is obvious that in the GPFS scheme, there is a direct relationship between the number of levels in the Data Flow Tree, the number of processing elements in the Sorting and Absorbing Units, the number of clauses in the given problems, and the communication bandwidth established between the Host and the GPFS. Pragmatically, to determine the optimal parameters for implementing the GPFS, the following procedure can be used:

(1) Examine the characteristics of the problems: determine the numbers of maximum allowable clauses in the GPF,

products per clause, literals per product, and percentage of negated literals.

(2) Generate the sets of simulation test cases: use program GPF-Generator (GENGPF) to randomly generate the GPF problems used for testing and evaluating different GPFS configurations.

(3) Determine the SAPA size: start with one level DFT (with a single BPP node) determine the SAPA size which produce the acceptable optimal solutions, with the smallest number of PEs, at an acceptable amount of execution time.

(4) Determine the number of DFT levels: increase the number of tree levels in the GPFS. Repeat the simulation to further reduce the SAPA size while still attain solutions at the same or lower costs (in comparison to those obtained in step 3).

(5) Analyze the processor utilization: adjust the number of DFT levels and the size of PMU local memory storage based on the efficiency of processor utilization, the total execution cycles required, and the Host's communication capability. Iterate steps 3 and 4 until the most optimized combination is achieved.

As a final illustration, let us consider the second problem in Table XI. Suppose that the Host is operating at 10 Mhz; in order to fetch all four leaf node BPPs in a three-level DFT in every execution cycle, the BPP's clock rate is set at 2.5 Mhz. With this setting, the GPFS can solve a 1000-

clause GPF in approximately 0.7 milliseconds (i.e., 1698 cycles x 0.4 milliseconds/cycle). On the other hand, a two-level GPFS, with 2 leaf node BPPs operating at 5 Mhz, can solve the same problem instance in 1.4 milliseconds (i.e., 3470 cycles x 0.4 milliseconds/cycle). To solve this problem instance by simulating the same scheme, a conventional computer, operating at 10 Mhz, requires 70.8 and 76.7 seconds, respectively.

CHAPTER V

EXTENSIONS TO THE GPFS ARCHITECTURE AND CONCLUSION

In the preceding chapters, the GPFS is introduced as a powerful tool to solve NP-hard combinatorial problems of logic design. The main design idea is to develop a special purpose computing system which is sufficiently efficient to solve large problems formulated into a Boolean expression, which must be general enough such that many other problems can be reduced to it. This thesis proposed a solution: the General Propositional Formula (GPF), and the corresponding GPF Solver (GPFS). As shown in the first two chapters, many logic design problems can be reduced to a Generalized Propositional Formula to be solved on the GPFS. Although this postulate is certainly applicable for many cases, in some particular circumstances, it might be desirable to obtain an additional tool to optimize the large clauses in the GPF in the Host, prior to submitting the problem to the GPFS. Or to capture the GPFS final output SOP expression, and derive a minimal subset of implicants (i.e., products in the final clause) which cover the entire set of literals existing in the original function. The Covering Problem Solver Algorithm is developed to fulfill this requirement.

THE COVERING PROBLEM SOLVER

The Covering Problem Solver (CPS) is designed as an extension to the GPFS. This algorithm, executed by the Host, finds the optimal cover of an SOP Boolean expression by performing the table reduction process, and by applying the tree search method to obtain an optimal cover set of implicants, where each implicant is a Boolean product of literals. Initially, the CPS examines the input data, computes the number of implicants, N_I, and number of distinct literals, N_L, in the given expression, then it encodes the input SOP expression into a switch table. Each literal of the given function is represented as a column, and each implicant (or product in the given function) is represented as a row in the table. With every data point in the table occupying by one bit in a computer word, the size of the switch table is N_I by N_L bits. In the switch table, the status of a bit (i.e., one or zero) at an intersection of a row and a column indicates the existence or absence of a certain literal (found in the current column) in a particular implicant (found in the current row). The CPS also translates the results into binary bitmap patterns, (i.e., switch dashes) using the above encoding scheme.

The CPS detects the domination relationship between two implicants, or two literals, which corresponds to rows or columns in the switch table. Based on the Quine-McCluskey

method of table reduction, the CPS iteratively detects and removes from the switch table all the essential columns, dominated rows, and dominating columns. Since the essential columns represent literals which are covered by only one implicant in the entire expression, to cover the whole set of literals, the corresponding essential implicants must be included to the final solution. By definition, [Kohavi, 1978], a row (column) is dominated by another if an one appears in any column (row) on the first row (column), and there also exists an one in the same column (row) on the second row (column).

The CPS shall dynamically allocate memory storage in the local memory unit to accommodate new input implicants (also called partial cover set), and deallocate memory storage to remove old partial cover sets when they are not further needed. Its main function is to generate a sequence of possible Absolute Optimal Cover from the reduced SOP expression. To accomplish this task, the CPS performs a Breadth First Search (BFS) through a tree, in which each child node (of cost j+1) at level j+1 of the tree being a combination of the parent node (of cost j) at the j-th level with another node of the same predecessor at the same level of the tree hierarchy. The CPS generates every node of the tree by mapping the rows in the reduced table and keeping track of the number of implicants in each combination as well as the number of literals that are included by the newly

created partial product. The first combination which satisfies the reduced table (i.e., has an one in every column) is a minimal cover set that contains the least number of implicants in it.

With a sophisticated computational methods, the CPS can efficiently accelerate the modified BFS process by:

- Recursively reducing the size of the tree based on the row domination property.
- Effectively producing new partial cover sets using intermediate results.
- Efficiently utilizing local memory storage by employing dynamic memory management techniques, and compact binary bitmap pattern representations for implicants and literals in the switch table.

Every time a partial cover set (a tree node) is generated, it is compared to the existing sibling nodes (i.e., nodes of the same parents). The dominated nodes are eliminated from the tree. As the number of predecessor (parent) nodes decreases, the numbers of successor (child) nodes and branches in the sub-trees of the succeeding levels is consequently reduced. New partial cover sets are formed based on the intermediate results. As a new node is generated by bitwise OR mapping of two sibling nodes from the previous level, a new partial cover set (of cost j) will be created by combining the two existing partial cover sets (of cost j-1), differentiated by just a single implicant. Following this modified BFS

method, the CPS searches thoroughly across every level of the tree. Once all the child nodes in the succeeding level are completely created and examined, all parent nodes in the previous level of the tree can be removed. The binary data format provides a compact and effective representation of implicants and literals in the switch table (i.e., the tabulated bitmap representations of the existence of a literal in an implicants; or an implicant which covers a group of literals). Since each implicant/literal is represented by only one bit in the appropriate switch dash (row or column in the switch table), new partial cover set can be generated by applying logically inclusive OR operation on the bitmap patterns. This method greatly reduces the size of the required run-time memory storage, and at the same time, cuts down the processing time. Finally, the CPS concatenates the optimal cover sets to the previously detected Essential Implicants to form the complete solutions.

Verification

Since the Covering Problem Solver is a sophisticated computational algorithm that requires variable length of execution time based on the data nature of the given problem, the CPS program has been developed to verify the efficiency and correctness of these eliminating and searching techniques. Further discussions regarding the CPS program, test cases, and results will be provided later in this chapter. In this

section, the two following examples shall provide us a concrete illustration to the process of searching for the optimal expression.

Example 1. Consider a SOP Boolean expression which contains 5 implicants (N = 5). The Covering Problem Solver perform the modified BFS tree search method to generate a sequence of all possible combinations of implicants: to form cover sets which include higher number of implicants. With 5 nodes in the top level of the corresponding N-ary tree, the total number of nodes in the tree is 31. Initially, five cover sets of cost 1 are generated, i.e., {A}, {B}, {C}, {D}, and {E}. These are simply cover sets of rows on the switch table. Next, the above cover sets are combined to form ten cover sets of cost 2: {A,B}, {A,C}, {A,D}, {A,E}, {B,C}, {B,D}, {B,E}, {C,D}, {C,E}, {D,E}. The cover sets of cost 2 are generated by bitwise ORing the two corresponding rows. Again, these cover sets can be combined (logically ORed) to produce eleven cover sets of cost 3: {A,B,C}, {A,B,D}, {A,B,E}, {A,C,D}, {A,C,E}, {A,D,E}, {B,C,D} ,{B,C,E} ,{B,D,E}, {C,D,E}. The process is repeated to construct five cover sets of cost 4 by combining the cover sets of cost 3: {A,B,C,D}, {A,B,C,E}, {A,B,D,E}, {A,C,D,E}, {B,C,D,E}. Finally, a cover set of cost 5: {A,B,C,D,E} is formed by combining all cover sets of cost 4. The process of generating the cover sets of cost j+1 from the existing cover sets of cost j is represented in Figure 9. The N-ary tree configuration is also illustrated.

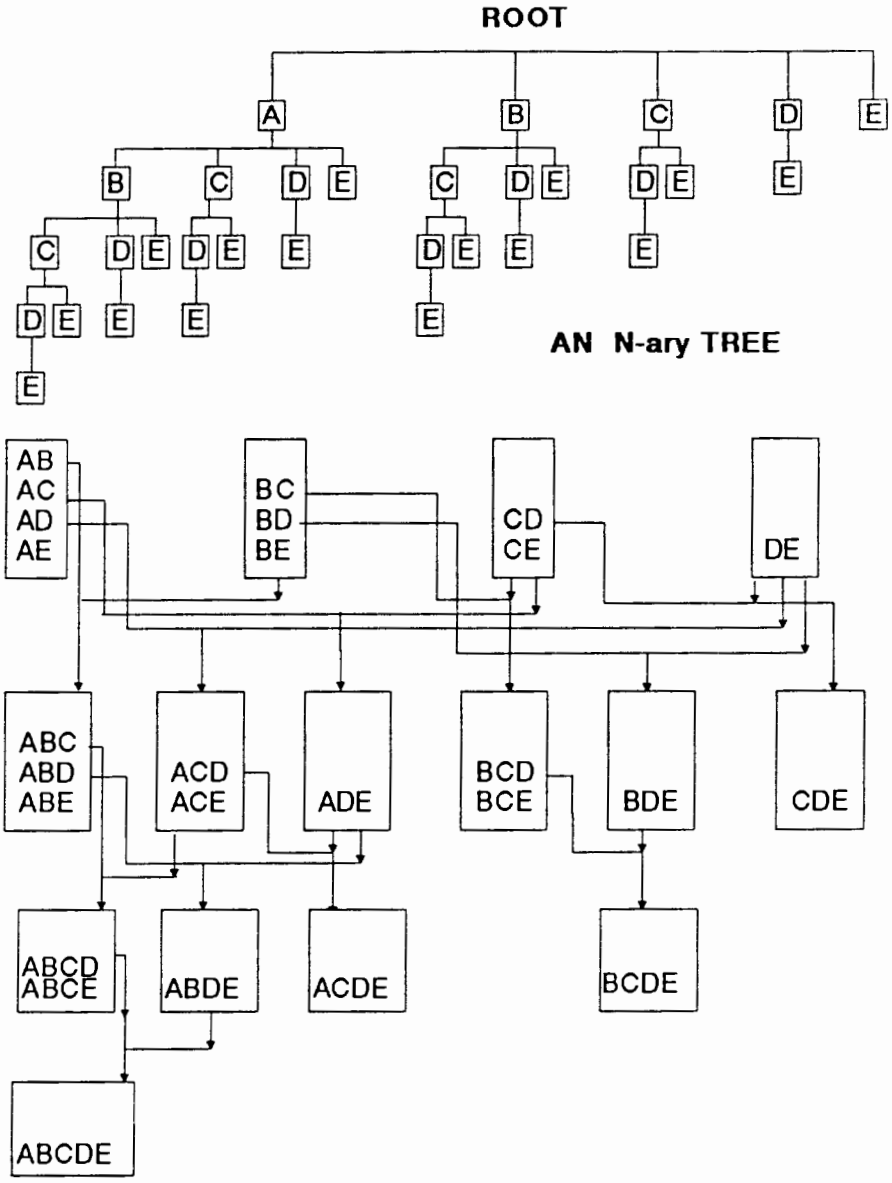


Figure 9. The breadth first tree search approach used in the covering problem solver algorithm.

Example 2. Given is the Boolean expression,

bgt + hnu + ckq + adghr + best + cfg +

c + ber + amp + bghku + enp + crs +

ahu + abfmq + e + b + cgn + ahmr

which contains 18 implicants (N_I = 18), and 17 literals (N_L = 17). In this example, the lower-case characters are used to represents literals, and upper-case characters are used to represent implicants. Thus, with this convention, the implicants are assigned as follows. A = bgt; B = hnu; C = ckq; D = adghr; E = best; F = cfg; G = c; H = ber; J = amp; K = bghknu; L = enp; M = crs; N = ahu; P = abfmq; Q = e; R = b; S = cgn; and T = ahmr. The corresponding switch table is shown in Table XII. First of all, the table reduction process recognizes column 4 (i.e., literal d) as an essential column. It eliminates the column and saves the associated implicant, D, to be included in the final optimal products. As column 4 is removed, row D is removed, and consequently, columns 1, 7, 8, and 14 are also removed from the table. Since no other essential implicant is found in the table at this point, the process continues on eliminating all dominated rows: rows A, H, and R are dominated by E; row N is dominated by K, row Q is dominated by L; and row T is dominated by P. Next, column 2 is removed since it dominates column 16. No other dominating column is detected. However, column 16 now becomes an essential column since the literal on that column, t, is now covered by only one implicant, E. The entire table reduction process is repeated. As column 16 is eliminated, columns 5, 15, and row E are also removed. Subsequently, row M is deleted for it is dominated by row F. The reduction process is completed. Table XIII is the reduced table. The

corresponding Boolean expression for, with number of implicants and literals, $N_I = N_L = 8$, is as follows,

$$nu + ckq + cf + mp + ku + np + fmq + cp$$

The union set of any cover set of implicants that satisfactorily covers the reduced table and the formerly detected set of essential implicants, {D,E}, is a solution to the original SOP expression.

At this stage, the simplified Boolean function can be solved as shown in Example 2. The modified BFS process is activated to generate all possible cover sets of implicants that satisfy the reduced table. The step-by-step operation of this process is illustrated in Table XIV. The internal bitmap representations of the combinations of all implicants in the generated cover sets are provided. The cover sets which contain ones in every location of its bitmap pattern, by ORing altogether the bitmap patterns of the implicants in the set, are solutions to the covering problem. Because the column orders are not important in solving this problem, by using this method, they are excluded from the table. The first solutions obtained are the cover sets of cost 4: {B,C,J,P}, {C,K,L,P}, {F,K,L,P}, and {K,L,P,S}. Since we are interested only in the cover sets of the least costs, the process can be concluded here. The final optimal cover sets (of cost 6), are: {B,C,D,E,J,P}, {C,D,E,K,L,P}, {D,E,F,K,L,P}, and {D,E,K,L,P,S}.

TABLE XII

THE CPS OPERATIONS - THE SWITCH TABLE

IMPs vs. LITs	a 1	b 2	c 3	d 4	e 5	f 6	g 7	h 8	k 9	m 10	n 11	p 12	q 13	r 14	s 15	t 16	u 17
A	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
B	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1
C	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0
D	1	0	0	1	0	0	1	1	0	0	0	0	0	1	0	0	0
E	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0
F	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
G	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
H	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
J	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
K	0	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	1
L	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0
M	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0
N	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
P	1	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0
Q	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
R	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0
T	1	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0

TABLE XIII

THE CPS OPERATIONS - THE REDUCED TABLE

IMPs vs. LITs	c 3	f 6	k 9	m 10	n 11	p 12	q 13	u 17
B	0	0	0	0	1	0	0	1
C	1	0	1	0	0	0	1	0
F	1	1	0	0	0	0	0	0
J	0	0	0	1	0	1	0	0
K	0	0	1	0	0	0	0	1
L	0	0	0	0	1	1	0	0
P	0	1	0	1	0	0	1	0
S	1	0	0	0	0	1	0	0

TABLE XIV

THE CPS OPERATIONS - THE PROCESS OF GENERATING COVER SETS

COVER SETS	BITMAP PATTERN	STATUS
{B,C}	10101011	
{B,F}	11001001	
{B,J}	00011101	
{B,K}	00101001	- {B,C}
{B,L}	00001101	- {B,J}
{B,P}	01011011	
{B,S}	10001101	
{C,F}	11100010	- {C,P}
{C,J}	10110110	
{C,K}	10100011	
{C,L}	10101110	
{C,P}	11110010	
{C,S}	10100110	- {C,L}
{F,J}	11010100	
{F,K}	11100001	
{F,L}	11001100	
{F,P}	11010010	
{F,S}	11000100	- {F,L}
{J,K}	00110101	
{J,L}	00011100	
{J,P}	01010110	
{J,S}	10010100	
{K,L}	00101101	
{K,P}	01110011	
{K,S}	10100101	
{L,P}	01011110	
{L,S}	10001100	
{P,S}	11010110	
{B,C,F}	11101011	-{B,C,P}
{B,C,J}	10111111	
{B,C,P}	11111011	
{B,C,S}	10101111	-{B,C,J}

COVER SETS	BITMAP PATTERN	STATUS
{B,F,J}	11011101	
{B,F,P}	11011011	
{B,F,S}	11001101	-{B,F,J}
{B,J,P}	01011111	
{B,J,S}	10011101	
{B,P,S}	11011111	
{C,J,K}	10110111	
{C,J,L}	10111110	
{C,J,P}	11110110	
{C,K,L}	10101111	
{C,K,P}	11110011	
{C,L,P}	11111110	
{F,J,K}	11110101	
{F,J,L}	11011100	
{F,J,P}	11010110	
{F,K,L}	11101101	
{F,K,P}	11110011	
{F,L,P}	11011110	
{J,K,L}	00111101	
{J,K,P}	01110111	
{J,K,S}	10110101	
{J,L,P}	01011110	
{J,L,S}	10011100	
{J,P,S}	11010110	
{K,L,P}	01111111	
{K,L,S}	10101101	
{K,P,S}	11110111	
{L,P,S}	11011110	
.		
.		
{B,C,J,P}	11111111	***
{C,K,L,P}	11111111	***
{F,K,L,P}	11111111	***
{K,L,P,S}	11111111	***
.		
.		

NOTES:
-{X,X}Removed,
Dominated by {X,X}.
*** Optimal Cover Sets.

THE CPS ALGORITHM

The CPS Program Description

The CPS program consists of three modules: TABCOV, TABRED, and TABBFS.

The TABCOV module includes the Program Sequencer (MAIN), and procedures for receiving and calculating the numbers of implicants (N_I) and literals (N_L) in the given SOP expression (Exam-Input-Data, and Get-Input-Data). Based on the values of N_I and N_L, the CPS data structure is created (procedures Create-CPS, and Init-CPS). This data record contains counters, and header pointers to a number of linked lists of Essential Implicants, Optimal Cover Sets, and elements of the switch table (Literals, and Implicants). The dimension of switch table is defined to be the number of implicants (row) by the number of literals (columns). The switch table is created to accommodate the input data, obtained from a file (procedures Get-Input-Data, Create-Table-Row, Create-Table-Column, Enter-Row, and Enter-Column). To maximize the utilization of memory storage, and speed up the program execution, the resolution of the switch table is limited to one bit. Thus, a byte storage can contain the information of an implicant (row) in 8 consecutive columns, or information of a literal (column) in 8 consecutive rows. In an IBM PC-AT, a bitwise AND/OR operation, performed on two unsigned integer values, will affects 16 rows or columns of the switch

table at the same time. Since these bitwise operations are performed repeatedly for the table reduction process, this encoding method significantly enhances the performance of the program. The TABCOV module also contains some utilities procedures used for reporting the results, such as procedures Switch-Box-Report, Optimum-Product-Implicant-Report, AOP-Finder-Report, and Architecture-Report.

The TABRED module consists of necessary routines for table reduction process: procedure Table-Reduction iteratively invokes other routines (i.e., Essential-Implicants-Remover, Dominated-Rows-Remover, and Dominating-Columns-Remover) to detect essential columns, dominated rows, and dominating columns, and reduce the size of the switch table accordingly. At the starting point of the table reduction process, the linked-list of essential implicants (if any) is created, with its header pointer contained in the CPS header structure (CPS.ESS_Ptr). Whenever an essential column is detected, an essential implicant node is created and is linked to the header pointer. The corresponding counter is incremented. On the other hand, anytime a dominating row, or a dominating column is detected, the Shrink-Row-Switch-Dash or Shrink-Column-Switch-Dash procedures will be called to shorten the size of the encoded bitmap pattern by one bit at the specified bit location, where the domination occurred. The column and row counters are updated accordingly.

The TABBFS module contains all procedures necessary to

perform the modified Breadth First Tree Search (BFS) to generate all possible optimal cover sets. Procedure SOL-Generator generates an N-ary tree with the number of nodes equals to the number of implicants (rows) remaining in the reduced table. Then it repeatedly invokes other routines in the module (i.e., Create-PMP-Node, Start-New-Finder-Level, Linking-New-Product-Node, and Remove-Previous-Finder-Level) to combine the nodes from the same tree level to create all possible child nodes located at the successive levels of the tree. Each product node will be correlated with the existing nodes from the same tree level for domination relationship. Since all product nodes in the tree are stored under linked-list structure in the local memory storage, they are easily removed from the tree. Also, since the product nodes (of higher costs) residing in the next level of the tree are created based merely on the combinations of the nodes in the current level, as all the child nodes of the next tree level are generated, the parent nodes of the current tree level may be removed. This program provides the users with a number of options to control the program execution, and to format the final report (CPS-Run-Options).

Test Cases and Results

To evaluate the CPS performance, a program has been written to randomly generate Covering Problems of various sizes for testing purposes (GENCOV). Table XV represents the

test results for a number of test cases.

The source listings of the CPS program, together with the test case generator, GENCOV, and the sample script file containing a test case and the corresponding test results can be found in appendices C, and D.

TABLE XV

THE COVERING PROBLEM TEST RESULTS

TEST No.	NUMBER OF			REDUCED TABLE		MIN. COVER SETS	EXEC TIME in SECs
	Implicant	Literal	Essential Implicant	Rows	Columns		
1	25	25	4	0	0	4	2.4
2	25	25	3	4	4	3	2.2
3	25	25	2	6	6	5	1.8
4	25	50	3	11	10	6	2.4
5	25	50	3	0	0	3	4.5
6	50	50	1	27	18	8	12.0
7	50	50	2	25	18	5	4.1
8	50	100	8	14	10	12	3.8
9	50	100	4	38	27	10	14.7
10	100	100	4	67	26	10	48.6
11	100	100	1	36	18	9	16.4
12	100	150	2	18	14	7	14.1
13	100	150	23	0	0	23	10.3
14	100	200	6	7	7	9	11.7
15	100	200	13	14	11	17	15.0

Through the test results, the Covering Problem Solver algorithm has been proven to be very efficient, and for many cases it was able to produce absolute optimal solution while solving expressions of small sizes (less than 100 implicants, and 100 literals). However, for special problems, such as in

the cyclic map, the table reduction process cannot further reduce the size of the switch table, the BFS process may require unjustifiable length of operating time, and significant memory space to generate, and to accommodate all possible combinations of the implicants of the final SOP expression. If we are interested not in the exact optimal solutions only, then the quasi-optimal solutions may be obtained by solving the problem employing various heuristic approaches such as the Branching Method or the Function Decomposition Method. In the first approach, an implicant (row) is selected, and removed from the table. This implicant will be treated as an essential implicant, and must be included to every product of the final expression. After the selected row is eliminated, the switch table might introduce new essential implicants, or dominating columns, or dominated rows to be further deleted. The reduction process shall continue on until no essential implicants, dominated rows, or dominating columns are detected. In cases of the size of the reduced table still too large to be solved in a reasonable length of time, the selection process might be repeated to reinitiate the table reduction method to shrink the table to a solvable size. In the Function Decomposition Method, the given table is partitioned into smaller tables, to be solved separately. The combinations of the resultant expressions constitute the final result.

FURTHER EXTENSIONS

The GPFS architecture can also be used for very fast sorting, sorting with absorbing, or absorbing of Boolean functions in the form of array of cubes. It can be also used to execute these operations for binary vectors, which finds applications to many problems. Replacing parallel operations on the long words with serial operations would significantly reduce the implementation cost of the architecture with only linear decrease in the execution speed. The generalization of this architecture can be done by extending the number of bit-by-bit operations in the Cartesian Product Generator from bitwise AND to include all 2-variable Boolean functions which has applications to other cube calculus operations. For instance, the GPFS can be extended to solve the Boolean expression of the form Product of "Exclusive-OR Sums" of Products of literals (where the literals may be in negated, or non-negated form).

CONCLUSION

As a wavefront architecture, the GPFS provides many desirable features of an array processor system, some of its major advantages can be described as follows [Kung, 1988].

(1) High-speed performance: The results of the simulation indicated that the GPFS provides a high speed performance which is adequate for many practical applications.

(2) Cost-effectiveness: The modularity of the GPFS make it very cost-effective and suitable for VLSI implementation. The number of module types in the system is limited to two: the Product Management Unit, and the Sorting and Absorbing Parallel Architecture. Since the operations performed by each of these module types are adequately general, they can be individually utilized for other applications with some minor modifications. Also, the regularity of the interconnections between the Boolean Product Processors in the Data Flow Tree, between the processing units in each Boolean Product Processors, and between the PEs in each Sorting and Absorbing Parallel Architecture allows the system to be theoretically extended indefinitely.

(3) Flexibility: The GPGS can be programmable and reconfigurable to maximize its performance efficiency for solving problems of different sizes.

Above all, the data-driven operation eliminates the need for an accurate global clock in the system; thus, frees the designers from dealing with critical timing problems. The flexible asynchronous scheme of the GPFS is perhaps the greatest advantage of this architecture.

REFERENCES

Brayton, R., G. Hachtel, C. McMullen, and A. Sangiovanni-Vicentelli, Logic Minimization Algorithms for VLSI Synthesis, Kluwer Academic Publishers, Boston, MA, 1984.

Brayton, R., "Minimization of Boolean Relations," Proc.22nd ISCAS, IEEE Circuits and Systems Society, Piscataway, NJ, 1989.

Bose, R. and B. Manvel, Introduction to Combinatorial Theory, John Wiley & Sons, New York, 1984.

Choi, Y. and M. Malek, "A Fault-Tolerant VLSI Sorter," Proc. ICCD, IEEE Computer Society, Long Beach, CA, 1985, pp. 510-513.

Cook, S., "The Complexity of Theorem-Proving Procedures," Proc. 3rd ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, 1971, pp. 151-158.

Demuth, H., "Electronic Data Sorting," IEEE Transactions on Computers, Vol. C-34, No. 4, April 1985, pp. 296-309.

Dietmayer, D., "Logic Design of Digital Systems", Allyn and Bacon, Boston, MA, 1971.

Garey M. and D. Johnson, Computers and Intractability: a Guide to the Theory of NP-Completeness, Freeman, San Francisco, CA, 1979.

Gonnet, G., Handbook of Algorithms and Data Structures, Addison-Wesley, Reading, MA, 1984, pp. 118-162.

Hayes, J., Computer Architecture and Organization, McGraw-Hill, New York, 1978.

Ho, P. and M. Perkowski, "Systolic Architecture for Solving NP-Hard Combinatorial Problems of Logic Design and Related Areas," Proc. 22nd ISCAS, IEEE Circuits and Systems Society, Piscataway, NJ, 1989.

Horowitz, E. and S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Rockville, Maryland, 1978.

Hu, T., Combinatorial Algorithms, Addison-Wesley, Reading, MA, 1982.

Jagadish, H., R. Matthews, T. Kailath, and J. Newkirk, "A Study of Pipelining in Computing Arrays," IEEE Transactions on Computers, Vol. C-35, No. 5, May 1986, pp. 431-437.

Jump, J. and S. Ahuja, "Effective Pipelining of Digital Systems," IEEE Transactions on Computers, Vol. C-27, No. 9, September 1978, pp. 855-862.

Jung, A. and K. Mehlhorn, Parallel Algorithms and Architectures, Springer-Verlag, Berlin, Germany, 1987.

Karp, R., "Reducibility Among Combinatorial Problems," in Complexity of Computer Computations, ed. by R. Miller and J. Thatcher, Plenum Press, New York, 1972.

Knuth, D., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.

Kohavi, Z., Switching and Finite Automata Theory, McGraw-Hill, New York, NY, 1978.

Kung, S., VLSI Array Processors, Prentice Hall, Englewood Cliffs, NJ, 1988.

Le, V., "A New General Purpose Systolic Array for Matrix Computations," M. Sc. Thesis, Electrical Engineering Dept., Portland State University, Portland, OR, 1988.

Mead, C. and L. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, MA, 1980.

Miller, R., "Writing SIMD Algorithms," Proc. ICCD, IEEE Computer Society, Long Beach, CA, 1985, pp. 122-125.

Miranker, G., L. Tang, and C. Wong, "A 'Zero-Time' VLSI Sorter," IBM Journal of Research and Development, Vol. 27, No. 2, March 1983, pp. 140-148.

Nassimi, D. and S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," IEEE Transactions on Computers, Vol. C-27, No. 1, January 1979, pp. 2-7.

Page, E. and L. Wilson, An Introduction to Computational Combinatorics, Cambridge University Press, London, Great Britain, 1979.

Perkowski, M., "Systolic Architecture for the Logic Design Machine," Proc. ICCAD, IEEE Computer Society, Long Beach, CA, 1985, pp. 133-135.

Perkowski, M. and J. Brandenburg, "Solving Basic Boolean Algebra Problems on a Hypercube Computer," Report, Electrical Engineering Dept., Portland State University, Portland, OR, 1989.

Ralston, A. and E. Relly, Jr., Encyclopedia of Computer Science and Engineering, Van Nostrand Reinhold, New York, NY, 1983, pp.1026-1029.

Sasao, T., "Input Variable Assignment and Output Phase Optimization of PLA," IEEE Transactions on Computers, Vol. C-33, No. 10, October 1984, pp. 879-894.

Sedgewick, R., Algorithms, Addison-Wesley, Reading, MA, 1984, pp. 513-534.

Shin, H., A. Welek and M. Malek, "I/O Overlapped Sorting Schemes for VLSI," Proc. ICCD, IEEE Computer Society, Long Beach, CA, 1983, pp. 731-734.

Stone, H., C. Tien, M. Flynn, S. Fuller, W. Lane, H. Loomis, W. McKeeman, K. Magleby, R. Matick, T. Whitney, Introduction to Computer Architecture, Science Research Associates, Chicago, Chicago, 1983.

Svoboda, A., "Parallel Processing in Boolean Algebra," IEEE Transactions on Computers, Vol. C-22, No. 9, September 1973, pp. 848-851.

Tagaki, N. and C. Wong, "A Hardware Sort-Merge System," IBM Journal of Research and Development, Vol. 29, No. 1, January 1985, pp. 49-66.

Thompson, C., "The VLSI Complexity of Sorting," IEEE Transactions on Computers, Vol. C-32, No. 12, December 1983, pp. 1171-1183.

Todd, S., "Algorithm and Hardware for a Merge Sort Using Multiple Processors," IBM Journal of Research and Development, Vol. 22, No. 5, September 1978, pp. 509-517.

Uhr, L., Algorithm-Structured Computer Arrays and Networks, Academic Press, Orlando, FL, 1984.

Varman, P. and I. Ramakrishnan, "Speeding Up Sorting on an Array Processor," Proc. ICCD, IEEE Computer Society, Long Beach, CA, 1985, pp. 604-606.

Winslow, L. and Y. Chow, "The Analysis and Design of Some New Sorting Machines," IEEE Transactions on Computers, Vol. C-32, No. 7, July 1983, pp. 677-680.

Yasuura, H., N. Takagi, and S. Yajima, "The Parallel Enumeration Sorting Scheme for VLSI," Proc. ICCD, IEEE Computer Society, Long Beach, CA, 1982, pp. 1192-1201.

Zakharov, V., "Parallelism and Array Processing," IEEE Transactions on Computers, Vol. C-33, No. 1, January 1983, pp. 45-57.

APPENDIX A

THE SORTING AND ABSORBING PARALLEL ARCHITECTURE

In Boolean algebra, the Absorption Law is an essential tool for simplifying a given expression by eliminating the redundant literals. Attempting to expand a large Boolean expression without applying the Absorption Law may quickly reach a condition known as "combinatorial explosion", in which the memory space and the time required to accommodate and process the elements in the expanded expression increase exponentially with respect to the size of the given problem. For example, it would take 4^{20} execution steps to completely expand a small Petrick function that contains 20 clauses, 4 literals in each, without applying the Absorption Law. Assume that the above problem is to be solved on a computer (operating at the speed of 20 Mhz) which can complete a multiplication in every single execution cycle, the necessary operating time and memory storage for this case is approximately 14 hours and 2 trillion bytes of memory, respectively. In conjunction with the Absorption Law (e.g., A + AB = A), other basic properties of switching algebra such as Idempotency (e.g., A + A = AA = A), Complementation (e.g., A + A' = 1, and AA' = 0), Consensus theorem (e.g., AB + A'C + BC = AB + A'C, and (A + B)(A'+ C)(B + C) = (A + B)(A'+ C)),

and so on, are great contributors to the process of eliminating redundant and dominating products during the function expansion operations. Nevertheless, in many problems with larger sizes as mentioned in the preceding chapters, it is hardly possible to obtain the exact optimal solutions by simply examining all possible products. With decomposition methods, the given expression can be partitioned into smaller problems to be solved in parallel, on multiprocessor systems. But to select the least cost products from these resultant expressions, sorting is definitely necessary.

Sorting has been a fascinating problem for many engineers, mathematicians and computer scientists for years [Demuth, 1985], [Thompson, 1983]. Since sorting is one of the most important operations in data processing (e.g., information searching / retrieving problems, compiler and assembler design, operating system development, database system organization, digital signal processing applications, knowledge storage retrieving and artificial intelligence decision making problems, etc.), faster sorting schemes are always in high demand [Knuth, 1973]. Many sorting algorithms have been developed and introduced during the last few decades. They fall into various classes of sorting schemes, ranging from single processor sequential machines [Gonnet, 1984] to multiprocessor parallel architectures. Advances in VLSI technology encourage the development of techniques for achieving a higher degree of parallelism in tightly-coupled

multiprocessor systems. Some of the well known hardware design techniques for utilizing parallelism to improve the reliability, performance, and efficiency of digital systems are concepts of pipelining [Jump et al., 1978], I/O overlapping, memory caching, memory interleaving, replicating processing elements, system redundancy, and multiprocessor systems [Stone, 1975], [Le, 1988]. Many parallel sorting schemes have been studied and published. They are widely different in many aspects of structural organization and dataflow direction: from mesh, tree, ladder, to linear array organization; and from parallel input - parallel output (PIPO), parallel input - sequential output (PISO), to sequential input - sequential output (SISO) sorting schemes [Winslow, 1983]. To name a few, the "Zero Time" VLSI Sorter [Miranker, Tang, and Wong, 1983], Quad Tree Sorter [Ho, Perkowski, 1989], Rebound Sorter [Chen, 1978], Parallel Enumeration Sorter [Yasuura, Takagi, and Yajima, 1982], Ripple Sorter [Shin, Welek, and Malek, 1983], and many others.

The Sorting and Absorbing Parallel Architecture (SAPA) is specifically designed to

(1) Receive Boolean products of literals, which are sequentially generated by other unit.

(2) Sort the products in the given SOP expression into descending order, according to the product costs.

(3) Mark all dominating and redundant products for later elimination process.

Since the I/O dataflow through the SAPA is serial, this unit must be based on a SISO parallel sorting scheme. This criterion facilitates the selection process: all PIPO mesh, and PISO tree sorting structures are excluded. Among the available SISO linear array and ladder structure parallel sorters, the Ripple Sorter appears to be the most suitable model for this application. The SAPA, based on the basic architectural concept of the Ripple Sorter, will be described in the following sections.

THE RIPPLE SORTER

The Ripple Sorter was designed and first introduced in a paper titled "I/O Overlapped Sorting Schemes for VLSI", by H. Shin, A. Welek, and M. Malek of University of Texas at Austin [1983]. In the paper, they described and compared this sorter with two other parallel sorting machines of the same class: The Rebound Sorter [Chen, Lum, and Tung, 1978], and the Parallel Enumeration Sorter [Yasuura, Takagi, and Yajima, 1982]. The provided comparison analysis revealed that this sorter outperformed its competitors in many aspects of the design: being constructed in linear array topology, with less processing elements (i.e., N/2 PEs), lower degree of connectivity in its PEs, and simpler control circuitry, this sorter approaches the speed of 2N cycles for sorting a sequence of N items. This required sorting time, the same as that of the Parallel Enumeration Sorter (which requires N

PEs), and half that of the Rebound Sorter (with N-1 PEs), made this sorting scheme very cost effective. The area-time product cost of the Ripple Sorter is only half that of the Enumeration Sorter, and approximately one fourth the cost of the Rebound Sorter.

Architectural Description

The Ripple Sorter (RS), as shown in Figure 10, is a pipelined, I/O Overlapped, Serial Input Serial Output (SISO) sorting system. It is organized in a linear array structure of cascaded systolic processing elements. The sorter consists of N/2 PEs. "Each PE includes two shift registers P[i], Q[i], a comparator, and a control circuit. The shift register P[i] (or Q[i]) is connected to P[i-1] (or (Q[i-1]), where 1 <= i <= N/2. Assume that a register P[N/2 +1] is always set to MAX, and let Q[0] and P[0] denote the input and output buffers, respectively" [Shin, Welek, and Malek, 1983]. In this description, **MAX** is the largest possible value, to which all registers in the Ripple Sorter are initialized.

The control circuit in each PE consists of two external (global) control signals X and RESET, and a 2x2 switch, which is controlled by the output of the comparator and signal X. The RESET signal is used to initialize all registers to MAX (all bits are set to 1). The X signal controls the directions of data flow (I/O) through the sorter. During the first N execution cycles, X is set to 1, the sorter is in the "input"

state and data are fetched into the sorter through Q[1]. For the next N cycles, X is reset to 0, the sorter outputs sorted items serially through P[1]. The data in registers Q[i] and P[i] are parallelly compared. The 2x2 crosspoint switch directs the data transfers and exchanges the contents of registers, if necessary, such that the smaller items in the PEs reside always in registers P[i] at the end of each execution step. During 2N execution cycles, with N/2 PEs operating concurrently, N^2 comparisons (and swaps) are performed.

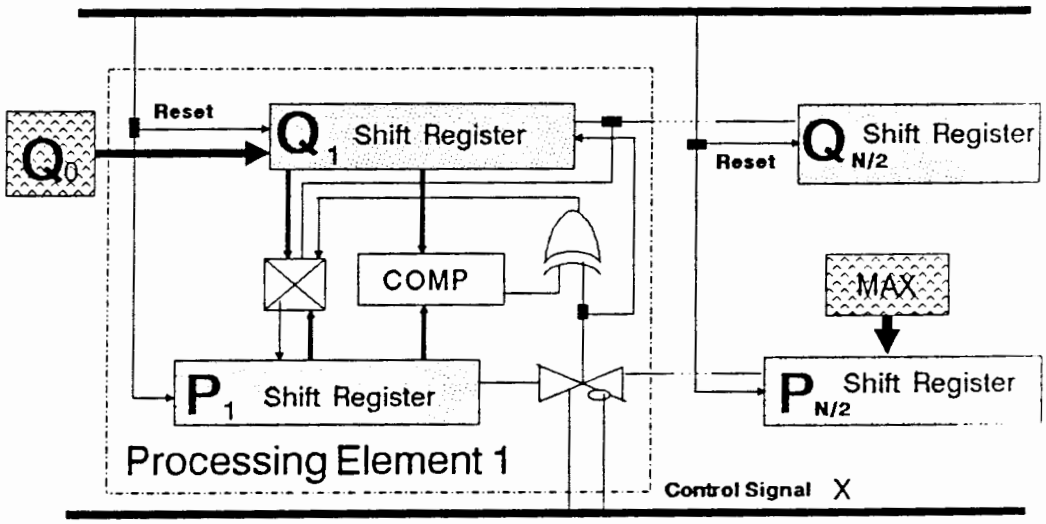


Figure 10. The ripple sorter organization.

Operational Description

The Ripple Sorter operation can be illustrated using the following algorithm in Parallel Pascal [Miller, 1985]. The entire structure of the Ripple Sorter can be visualized as a

parallel array of PEs. The main characteristic which distinguishes a parallel array from a regular array (as in standard Pascal) is as follows. For each executed instruction which accesses the array, all components of the parallel array, within a specified index range, are operated concurrently at every execution cycle. In contrast, in a regular array, the operation is sequentially performed by only one element of the array at a time.

```
PROCEDURE RIPPLE_SORTER;
BEGIN

     (** Initialization Phase: System RESET. 1 cycle. **)
     FOR i:=1 TO N/2 PARALLELLY DO BEGIN
          P[i] := MAX;
          Q[i] := MAX;
          END;

     (** Phase I: INPUT unordered sequence. N cycles. **)
     FOR i:=1 TO N DO BEGIN
          FOR j:=1 TO N/2 PARALLELLY DO BEGIN
               Q[0] := Input_Datum;
               Q[j] := Q[j-1];
               IF (Q[j] < P[j]) THEN BEGIN
                    P[j] := Q[j]; (* Swap *)
                    Q[j] := P[j];
                    END;
```

```
                END;
            END;

        (** Phase II: OUTPUT ordered sequence. N cycles. **)
        FOR i:=1 TO N DO BEGIN
            FOR j:=1 TO N/2 PARALLELLY DO BEGIN
                IF (Q[j] < P[j]) THEN BEGIN
                    P[j] := Q[j]; (* Swap *)
                    Q[j] := P[j];
                    END;
                P[j] := P[j+1];
                P[0] := P[1]; (* Output_Datum *)
                END;
            END;
    END; (** Procedure RIPPLE_SORTER **)
```

THE SAPA

Every product in the Boolean SOP expression sent to the SAPA is encoded as a bitmap pattern, which has the same dimension (i.e., number of bits) as that of the Q[i] (or P[i]) registers. The internal organization of each bitmap pattern (encoded word) can be partitioned into three fields: a Tag Field, a Cost Field, and a Product Field.

(1) The Tag Field - the single Most Significant Bit (MSB) of the encoded word - is used to determine if the product will be included in the final solution set. A value of

zero in this bit location indicates that the corresponding product is a dominating or redundant product, and shall be rejected from the final SOP expression. For instance, in the expression ABCD + AD + CE + AE + AD, the second AD (redundant product) and the ABCD products (dominating product) are eliminated. A dominating product contains a larger set of literals, which includes all the literals in the corresponding dominated product as a subset of it. Since the final expression is in SOP form, the elimination of the dominating products does not affect the search for a minimal solution. Moreover, redundant products can be observed as a special case of dominating products: the same principle can be applied for them.

(2) The Cost Field - a group of bits following the Tag Field in the encoded word - is the unsigned binary representation of the number of literals contained in the associated product (e.g., the number of 01, and 10 patterns in the Product Field of a GPFS word). The minimum number of bits required for the Cost Field is Log_2N, where N is the maximum number of literals allowed in the original function.

(3) The Product Field - the remaining Least Significant Bits (LSB) in the encoded word - is the bitmap pattern representation of a product according to the applied encoding scheme.

As mentioned in the preceding section, the Ripple Sorter exhaustively compares every pair of data items in the given sequence (i.e., N^2 comparisons performed on each sequence of N items). An additional combinational circuit can be implemented for every PE of the Ripple Sorter to detect the domination relationship of the products in the given SOP sequence, contained in the registers P[i] and Q[i], at the same time a comparison is performed. Anytime a dominating product is recognized from a pair of compared products, its tag bit is reset to zero. As products in the entire sequence are tested against each other for domination relationship, and are sorted into descending order, the products with smallest costs are outputted first. At the output stage, the tag bit of the outputted products are examined sequentially and the products with a zero tag bit are excluded from the final SOP expression.

Architectural and Operational Descriptions

In the SAPA, all I/O sequences, operations, and control circuitry of the sorting portion are unmodified. However, as shown in Figure 11, to realize the SAPA, each processing element of the Ripple Sorter includes an additional Product Domination Detector (PDD). This unit is connected to the Tag and Product Fields bit locations of the shift registers P[i], and Q[i]. While the bitmap patterns are treated as entities in data transfers among the registers in the system, they are

separately processed by different processing units in a PE. The comparator in PE[i] compares the costs (Cost Field) of a pair of products stored in registers P[i] and Q[i], according to the algorithm described in the previous section. At the same instant, the PDD checks the domination relationship of the two products by bitwise OR mapping the two encoded patterns (Product Field), and logically exclusive ORing the resultant with each of the original patterns. The resultant bit patterns from the bitwise XOR operation (called XOR-patterns) will determine the domination relationship of the two products of interest. An XOR-pattern with all zeros indicates that the associated product is dominated by the other product: the PDD, consequently, clears the tag bit (Tag Field) of the second product. If no XOR-pattern with all zeros is detected: no domination occurs; both products will have their tag bit set to one. If both XOR-patterns contain all zeros in all locations of their Product Fields, a redundancy occurred, the first product will have its tag bit reset by default. Since the results of the PDD only affect the status of the tag bit, the directions of data transfers among the registers in the SAPA will be determined by results of the comparisons.

The PDD, as shown in Figure 12, is a linear array of M combinational circuits. M is the number of bits in the Product Field. Each combinational circuit is associated with a particular pair of bits in the same bit position of

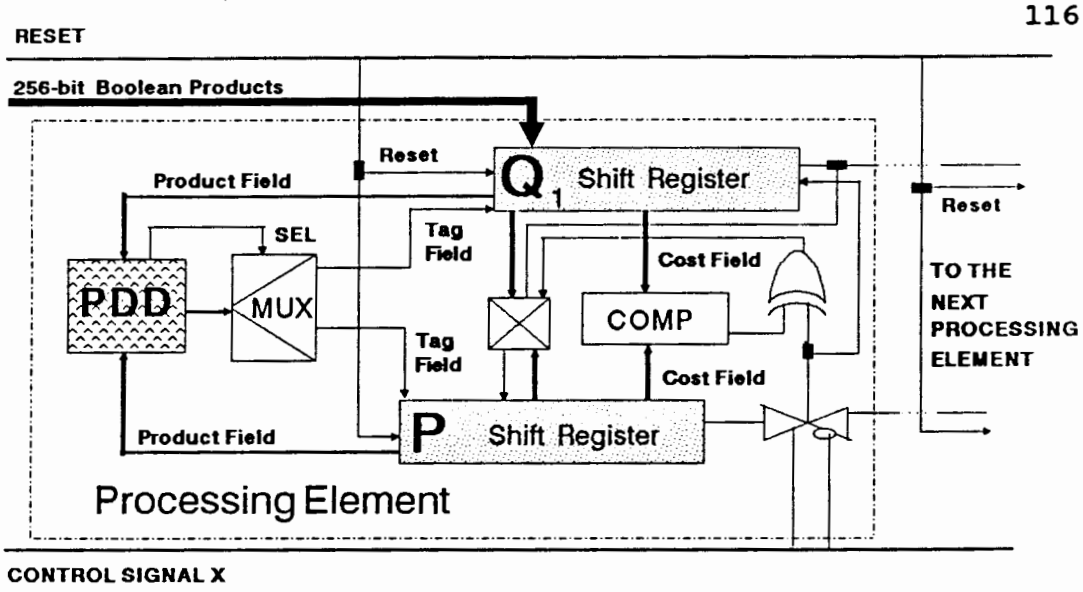


Figure 11. The sorting and absorbing parallel architecture.

registers P[i] and Q[i]. The PDD performs the bitwise OR, bitwise XOR operations on the bits in the Product Field of P[i] and Q[i] as described in the previous paragraph. To check the results of the XOR-patterns, the outputs of the XOR gates in each group are hardwire-ORed together. These lines, together, indicate the results of the domination check: combinations 00 and 01 indicate that the product in Q[i] is the dominating product. Combination 10 signifies that the contents of P[i] is the dominating product. The remaining combination 11 indicates that neither of the two products is a dominator. Consider, for example, the detection of the domination relationship of the two products ACE (stored in Q[i]), and AC (stored in P[i]). For the sake of simplicity, let us assume that the number of variables in the given function is five (e.g., A, B, C, D, E) and no negated variable

is allowed. The Product Fields in the Q and P registers, Q[i, Product Field] and P[i, Product Field] are, respectively, 10101 and 10100. By logically inclusive ORing the two bit patterns, we obtain 10101. Again, by logically exclusive ORing the resultant pattern with Q[i, Product Field], and P[i, Product Field], separately, the two bit sequences 00000 and 00001 (i.e., XOR-patterns) are obtained. Next, by individually bitwise ORing all the bits in each of these bit patterns, the result of the domination check is combination 01, which indicates that the product in Q[i] dominates the one in P[i]. Consequently, the tag bit in Q[i] must be cleared. A simple hardware realization of the PDD, and the descriptions of the decoding scheme applied on the resultant combinations are presented in Figure 12.

The SAPA algorithm, expressed in parallel Pascal Pseudo-code, can be found in chapter III. From the Ripple Sorter algorithm, the following modifications were made:

(1) The product cost comparisons are performed based on the contents of the Cost Fields.

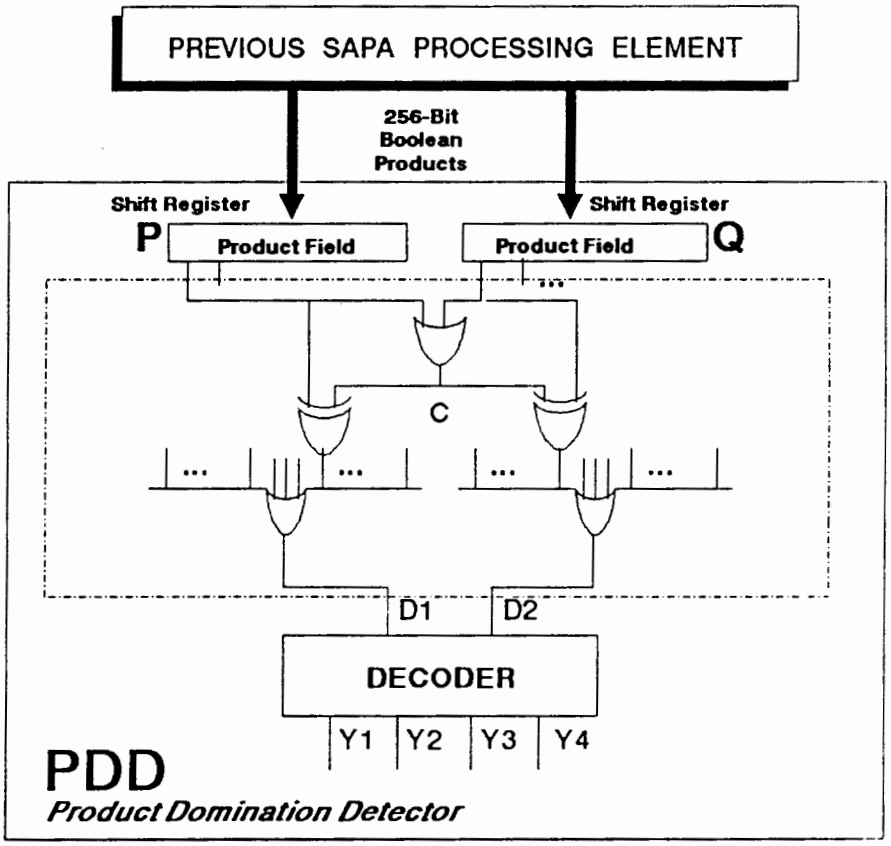
(2) The domination check procedure is done simultaneously on the Product Fields of the same pairs of products. The results may affect the tag bit.

(3) During the output phase (i.e., when the control signal, X, is reset to 0), every product outputted from the SAPA will be examined. Dominating or redundant products (with tag bit cleared) will be rejected.

The validity of this sorting and absorbing scheme is verified by means of examples and simulation. The following example should be sufficient to clarify the SAPA operations in sorting and absorbing a sequence of six Boolean products.

Example. Figures 13 and 14 illustrate a sequence of operations executed by the SAPA in rearranging and simplifying a SOP Boolean expression. Consider the expression: (A + BD) (D + BE + CH). By multiplying out the expressions in parentheses, the following Boolean products will be generated and sequentially fetched into the SAPA: AD + ABE + ACH + BD + BDE + BCDH. Since the given expression contains six variables, these products can be encoded in a nine bit pattern, with the MSB (bit 8) reserved for the Tag Field, three following bits (7, 6, and 5) reserved for the Cost Field, and the remaining four LSBs used for encoding the products. In this example, without any negated literal, each bit location can be assigned to a particular variable in the Product Field: bits 0, 1, 2, 3, 4, 5 are assigned to A, B, C, D, E, and H, respectively. A zero (or one) in a particular bit location signifies the presence (or absence) of the corresponding literal in the encoded product. Initially, all bits are set to one. Using the above encoding scheme, this sequence of products is sent to the SAPA as shown in Figure 13. The first N = 6 execution steps completely fill up all registers in the N/2 = 3 PEs of the SAPA. During the next N

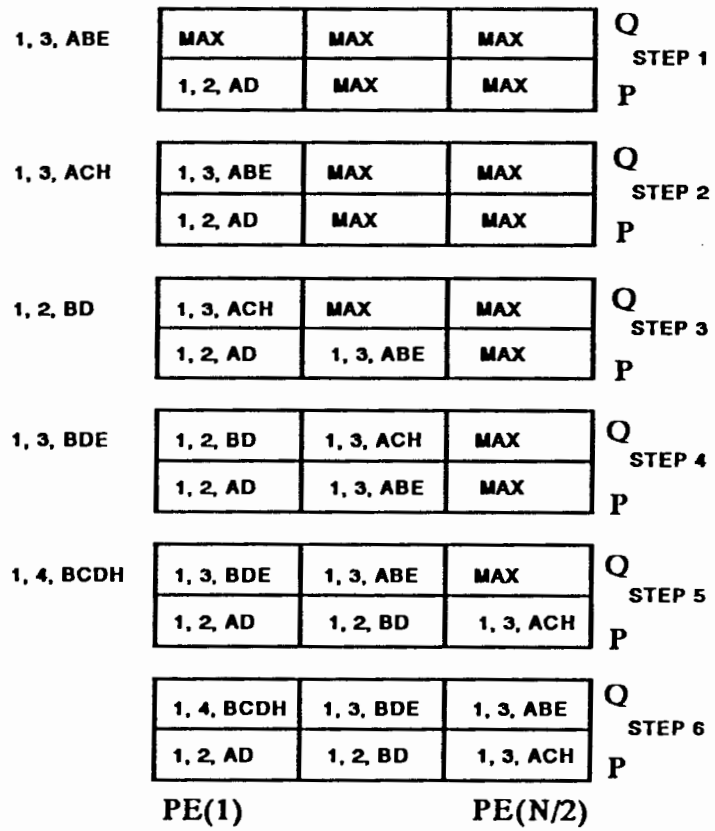
steps, the N given products are sequentially outputted, with the smallest cost product appearing at the output first. As illustrated in Figure 14, the order of products in the output sequence is AD, BD, ACH, ABE, BDE, and BCDH. BDE and BCDH



D1	D2	Y1	Y2	Y3	Y4	RELATIONSHIP	ACTIONS ROW	ACTIONS COLUMN
0	0	1	0	0	0	A is equal to B	Delete A	Delete A
0	1	0	1	0	0	A dominates B	Delete B	Delete A
1	0	0	0	1	0	A is dominated by B	Delete A	Delete B
1	1	0	0	0	1	A is not equal to B	Skip, Continue	Skip, Continue

Figure 12. The product domination detector.

have their tag bit set to zero, and are excluded from the final SOP expression since they dominate some other products in the sequence. To facilitate the verification process, the I/O products are shown in the format of X,X,XXXX, where the first field presents the status of the tag bit (0 or 1),



SAPA Input Phase:

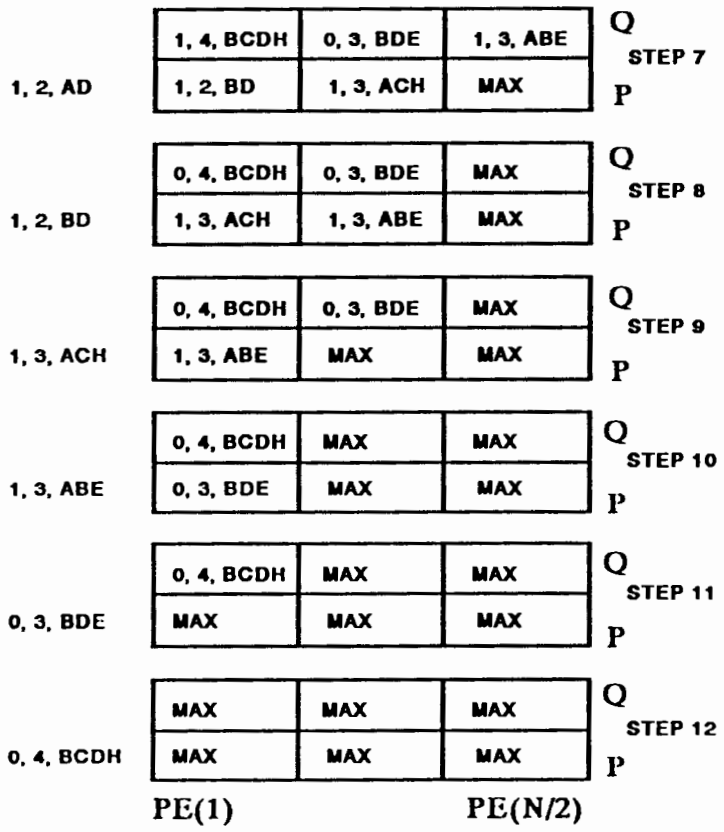
Q(i) = Q(i - 1)
If Q(i) < P(i) then
Swap P(i) and Q(i)
If Domination Occurs between P(i) and Q(i) then
Mark the Tag bit of the Dominating Product.

X, X, XXXX
Tag Cost Product

Tag bit=0: Dominating Product.
(to be discarded)

Figure 13. The SAPA operations - input phase.

the second field shows the cost of the product, and the third field contains the literals present in the encoded product. Alphanumeric characters are used in the illustrations instead of binary representation.



SAPA Output Phase:

If Q(i) < P(i) then
Swap P(i) and Q(i)
If Domination Occurs between P(i) and Q(i) then
Mark the Tag bit of the Dominating Product.
P(i) = P(i + 1)
P(n/2) = MAX

X, X, XXXX
Tag Cost Product

Tag bit=0: Dominating Product (to be discarded)

Figure 14. The SAPA operations - output phase.

REMARKS

The SAPA - a new parallel architecture for simplifying SOP Boolean expressions - is introduced. This sorting and absorbing hardware scheme is developed from a well-known parallel sorting machine: The Ripple Sorter.

The flexibility of solving size-independent problems is an essential characteristic which made the SAPA architecture more attractive and effective for numerous applications that require massive data processing. The SAPA can perform sorting and absorption operations on large SOP Boolean expressions which contain an arbitrary number of products without being reconfigured. By decomposing a long input sequence into shorter sequences which fit the existing SAPA dimension (i.e., the number of PEs in the system), the data in the partitioning sequences are being sorted and kept separately. Since the number of products in the input sequences are usually reduced based on their domination relationships, these shorter sequences can be combined and fed back to the SAPA to be reprocessed. The ability of solving size-independent problems allows the system designer to make a trade-off between system execution speed and the system realization cost. Additionally, since the SAPA is a serial input - serial output I/O overlapped architecture, in which the input and output items can only be obtained from the first element of its linear array of PEs, the system overhead is always N cycles,

where N is the number of items in the data sequence to be processed. As an example, a large SAPA of 500 PEs, configured to sort/absorb a long input sequence (e.g., 1000 items in 2000 execution cycles) can also be used, without modification, to process a much shorter sequences (e.g., 20 items) in a proportional length of time (40 execution cycles). In other words, although the SAPA provides the capability to massively pipeline and process large sequences of data items, it does not require filling up the pipelines to obtain the first output item.

The serial output characteristic of the design provides more flexibility for sequentially inspecting, and individually manipulating every data element in the sorted output list to meet any specific application requirements. As discussed in the next chapter, in the GPFS application, this characteristic allows the SAPA to reject all dominating (marked) products while sorting a sequence of Boolean products, and reset the system immediately after the first K optimal products are obtained.

The short and regular interconnections between the PEs allows the use of a high speed system clock. The simple, regular pattern of the SAPA basic processing element circuit, together with the systolic configuration of the SAPA structure, and the high degree of concurrency of the SAPA algorithm ensure its feasibility for very high scale integrated circuit (VLSI) implementation [Mead and Conway,

1980]. Further extensions to the Ripple Sorter include additional fault-tolerant features, proposed by Y. Choi and M. Malek in "A Fault-Tolerant VLSI Sorter" [1985]. The implementation of this technique may provide higher reliability to the SAPA results.

The SAPA simulator: Since the SAPA is designed to be used as a processing unit in the GPFS, a larger architecture, the SAPA simulation program is combined with other modules of the GPFS simulation programs (PMU.C, GPF.C, and HOST.C) to build the complete GPFS simulator. Module SAPA.C can be found in Appendix B.

APPENDIX B

THE GPFS ALGORITHM - PROGRAM LISTING

```
/*-----------------------------------------------------*/
/* GENERALIZED PROPOSITIONAL FORMULA SOLVER - Part#1 */
/*-----------------------------------------------------*/
/* LINK LIST VERSION.                                  */
/* This program simulates the operation of the GPFS. */
/* Version 2.0: 256 bit word with [Tag,Cost,Product] */
/*         (number of bits/field)   1    7    248    */
/*-----------------------------------------------------*/
/*     Filename:        HOST.C                         */
/*     Created:         FRI  02 JUN 1989       20:00 */
/*     Last Edited:     SAT  14 OCT 1989       12:00 */
/*-----------------------------------------------------*/


#include "GPF.H"
#include "TIMER.H"


/**********    E X T E R N A L  &  P U B L I C    *********/

Literal Literal_Arr[500];
Literal Input_Literal;
int Report_Frequency;
FILE *fopen();
FILE *fp;

Boolean Normal_Termination;
Boolean DEBUGS;

#include "SAPA.C"
#include "PMU.C"
#include "GPF.C"


/*************      P R O C E D U R E S      *************/

int Product_Decoder(Lits_P, Patt_P)

/* Decoding the given product nodes for displaying
   as SOP final results. Literals are stored in an
   array, pointed to by Lits_P.  The function
   returns the total number of negated/nonegated
   literals in the array.
*/
L_Ptr Lits_P; /* Pointer to an array of literals.
      E.g., Literals range from 1 to 124 [+/-],
      corresponding to bit locations 0 to 247
      [2 bits per literal]. NOTE WELL: Bits 248-254
      define the Cost field, 255 is the Tag bit. */
E_Ptr Patt_P;
{
```

```
  int i, j, k;
  Literal L;
  Element AWord, BWord;
  int Num_Lits = 0; /* Num of literals in decoded product */
  i = j = k = 0;
  while (i<WORD_SIZ) {
     AWord = *(Patt_P+i); /* Get an Element */
     j = 0;
     while ((j<ELT_LEN) && (k<MAX_LITS) &&
            (Num_Lits<MAX_LITS)) {
        BWord = AWord & 0x0003;
        if (BWord != 0x0003) { /* Not a Don't Care */
           L = (i * ELT_LEN) + j +1;
           if (BWord == 0x0001)
              L = -L; /*  negated literal "01" */
           /* else if (Bword==0x0002)
              ...non-negated literal "10" */
           *(Lits_P+Num_Lits) = L;
           Num_Lits++;
           }
        AWord = AWord >> 2;
        j++;
        k++; /* Number of checks */
        } /* while j & Num_Lits */
     i++;
     } /* while i */
   return(Num_Lits);
}  /* End Product_Decoder */


void Product_Encoder(Num_Lits, Lits_P, Patt_P)

int Num_Lits; /* Num of literals in encoded product */
L_Ptr Lits_P; /* Pointer to an array of literals */
E_Ptr Patt_P; /* Pointer to the product node pattern,
                 which contains the newly encoded
                 pattern (Prod_Node->Prod_Patt) */
{
  int i, j, m, n;

  for (i=0; i<Num_Lits; i++) {
     j = *(Lits_P+i);
                  /* Get one literal from the array */
     m = (j<0) ? -j : j;/* Index to the bit pattern */
     n = (m-1)%ELT_LEN; /* Location to be mapped
                           in each 16 bit quantity */
     m = (m-1)/ELT_LEN; /* Location of the 16 bit
                          quantity in the long word */
     /* Logically ANDing it to the pattern
     */
     if (j<=0) /* Negated literals: 01 */
```

```
        *(Patt_P+m) &= NEG_LITS[n];

      /* Different mask for non-negated literals: 10
         Logically ANDing it to the pattern
      */
      else  *(Patt_P+m) &= POS_LITS[n];
      }
}  /* End Product_Encoder */



void Print_Line (chr1, chr2, Len)
/*
   This function prints a line of length 'Len',
   consists of the given character 'chr2'. The
   function also allow some blank lines to be
   inserted, if desired, by sending a newline
   character for the 1st passing argument 'chr1'.
   The line length is adjusted accordingly.
*/
int chr1, chr2, Len;
{
    int i, j;
    for (i=0; i<3; i++) putchar(chr1);
    j = (chr1==chr2)  ? Len-4  : Len;
    for (i=0; i<j; i++) putchar(chr2);
    putchar(chr1);
}



void Print_Legends()
{
   void Print_Line();

   printf("\n(1) BPP#, (2) BPP Mail Box,");
   printf("(3) BPP Status,\n(4) Number of");
   printf(" Clauses, (5) Number of Products,");
   printf("\n(6) Number of idle cycles, (7)");
   printf(" CPG Status,\n(8) CPG Cnt,(9) ");
   printf("Sapa Switch,(10) SapaLeft Empty,");
   printf("\n(11) SapaRight Empty, (12) Sapa");
   printf("Left Input Cnt,\n(13) SapaLeft ");
   printf("OutCnt, (14) SapaRight Input Cnt,");
   printf("\n(15) SapaRight OutCnt,(16) Num");
   printf(" Clause Received.\n");
   Print_Line('\n','#',74);
} /* End Print Legends */
```

```
void GPFS_Report()
{
   int i, j;
   void Print_Line();

   printf ("\n-----------------------------------");
   printf ("-----------------------------------");
   printf ("\nExecution Cycle Number: %5d",
            HOST.Total_Exec_Cycles);
   printf ("\n----------BPP----------  --CPG--  ");
   printf ("------------SAPA---------------  NumC");
   printf ("\n i Mail Sta #Cl #Pr  #Idl Sta Cnt  ");
   printf ("Sw EmpL EmpR InL  OutL InR  OutR #Recv");
   printf ("\n(1) (2) (3) (4)  (5)  (6) (7) (8) (9)");
   printf (" (10) (11) (12) (13) (14) (15) (16)");

   for (i=0; i<NUM_BPPS; i++) { /* Performance Report */
      j = (GPFS[i].PMU->Rcvr_Ptr1 == NULL) ?  0 :
           GPFS[i].PMU->Rcvr_Ptr1->Num_Prods;
      printf
      ("\n%2d%4d%4d%4d%5d%5d%4d%4d%4d%5d%5d%5d%5d%5d%5d%5d",
                i,
                GPFS[i].Mail_Box,
                GPFS[i].Status,
                GPFS[i].PMU->Num_Clau_Possess,
                j,
                GPFS[i].Num_Idles,
                GPFS[i].PMU->CPG_Status,
                GPFS[i].PMU->CPG_Count,
                GPFS[i].SPRU->Switch,
                GPFS[i].SPRU->SAPAL->EMPTY,
                GPFS[i].SPRU->SAPAR->EMPTY,
                GPFS[i].SPRU->SAPAL->In_Cnt,
                GPFS[i].SPRU->SAPAL->Out_Cnt,
                GPFS[i].SPRU->SAPAR->In_Cnt,
                GPFS[i].SPRU->SAPAR->Out_Cnt,
                GPFS[i].Num_Clauses_Received);
      }
   printf("\nHOST STAT: %2d, MAILBOX: %3d",
      HOST.STATUS, HOST.Mail_Box);
   printf(", CLAUSES POSSESS: %2d",HOST.Num_Clau_Possess);
} /* End GPFS Report */



void Echo_a_Product(P_Ptr)
Prod_Ptr P_Ptr;
{
   int j,k,n;
   int Product_Decoder();
```

```
   /*  Print the list of literals in the given product
   */
   if (P_Ptr != NULL) {
      j = Product_Decoder(Literal_Arr, P_Ptr->Prod_Patt);
      printf("\n\t");
      for (k=0, n=0; k<j; k++, n++) { /* print them out */
         printf("%4d", Literal_Arr[k]);
         if (n%12 == 11) { /* 12 literals on a line */
            printf("\n\t");
            n = 0;
            } /* Print 12 literals on a line */
         } /* Echo a product */
      printf("\n\t[Product_Cost=%4d]",j);
      }
} /* End Echo_a_Product */



void Echo_a_Clause(C_Ptr)
Clau_Ptr C_Ptr;
{
   Prod_Ptr Pp;
   void Echo_a_Product();

   /*  Print the list of products in a given clause
   */
   if ((C_Ptr == NULL) || (C_Ptr->Num_Prods == 0))
      printf("\nEmpty Clause !!!");
   else {
      Pp = C_Ptr->Next_Prod;
      while (Pp != NULL) {
         Echo_a_Product(Pp);
         Pp = Pp->Next;
         } /* while not end of clause */
      printf("\n[Num_Products =%3d]",C_Ptr->Num_Prods);
      } /* else */
} /* End Echo_a_Clause */



void GPFS_Final_Report()

/*  Print the list of products in the final SOP
    expression. Print the statistical report of
    the BPPs for performance analysis purposes.
*/
{
   int i,j,k;
   int Num_Proc, AccI, AccR, AccX;
   Boolean Have_Solution;
   void Print_Line();
```

```
Print_Line('\n','#',74);
if ((HOST.Rcvr_Ptr != NULL) &&
    (HOST.Num_Final_Prods != 0) &&
    (Normal_Termination == TRUE)) {
   printf ("\nFINAL SOP SOLUTION:");
   Echo_a_Clause(HOST.Rcvr_Ptr);
   Have_Solution = TRUE;
   Print_Line('\n','#',74);
   } /* Echo contents of the final clause */
else {
   Have_Solution = FALSE;
   printf("\nNO PRODUCT IN THE FINAL EXPRESSION");
   }
printf ("\n\nFINAL REPORT ON BPPS PERFORMANCES.");
for (i=NUM_BPPS-1; i>=0; i--) {
/* Statistical Data Report
*/
   printf ("\nBPP#%2d:", i);
   printf ("\n\tNumber of IDLE cycles = %5d",
           GPFS[i].Num_Idles);
   printf ("\n\tNumber of Clauses Possess = %4d",
           GPFS[i].PMU->Num_Clau_Possess);
   printf ("\n\tNumber of Clauses Tranfer = %4d",
           GPFS[i].PMU->Num_Clau_Transfer);
   printf ("\n\tNumber of Clauses Received = %4d",
           GPFS[i].Num_Clauses_Received);
   }

Print_Line('\n','#',74);
printf("\nSUMMARY OF EXECUTION RESULTS:\n");
printf("\nDFT  IDLE  Receive  Transfer IDLE/ACTIVE");
for (j=0; j<NUM_DFTS; j++) {
    Num_Proc = (j==0) ? 1 : Num_Proc*2;
    AccI = AccR = AccX = 0;
    for (i=0; i<Num_Proc; i++) {
       k = i + Num_Proc-1;
       AccI+=GPFS[k].Num_Idles;
       AccR+=GPFS[k].Num_Clauses_Received;
       AccX+=GPFS[k].PMU->Num_Clau_Transfer;
       }
    AccI /= Num_Proc;
    AccR /= Num_Proc;
    AccX /= Num_Proc;
    printf("\n%2d%7d%8d%8d%12.2lf", j,AccI,AccR,AccX,
       (((double) AccI * 100.0 ) /
        (double) HOST.Total_Exec_Cycles));
    }

printf("\nHOST: Clauses returned from the DFT:%4d",
          HOST.Num_Feedback_Clauses);
if (Have_Solution == TRUE)
```

```
      printf("\nOptimal Product: Cost = %4d",
          Product_Decoder(Literal_Arr,
          HOST.Rcvr_Ptr->Next_Prod->Prod_Patt));
   else printf ("\nOptimal Product: NONE.");
   printf ("\nTotal GPFS Execution Cycles: %5d",
                        HOST.Total_Exec_Cycles);
} /* End GPFS Final Report */


void Release_a_Clause (Clause_P)
Clau_Ptr Clause_P;
{
   Prod_Ptr p, q;
   void Release_Product_Node();

   if ((p = Clause_P->Next_Prod) != NULL) {
      while (p->Next != NULL) {
         q = p->Next;
         Release_Product_Node((Prod_Ptr) p);
         p = q;
         }
      Release_Product_Node((Prod_Ptr) p);
      }
   free( (Clau_Ptr) Clause_P);
} /* End Release_a_Clause */


Clau_Ptr Create_Clause_Node ()

/*  Allocate memory for a Product Node which contains
    2 pointers, pointing to a Product node, and a
    neighbor clause node, respectively.
*/
{
     int i;
     Clau_Ptr C_Ptr;

     if ((C_Ptr = (Clau_Ptr) malloc
                  (sizeof(Clause_Node))) == NULL) {
        printf("\nErrors in creating clause nodes.");
        exit(11);
        }
     /* Initialize the Clause Node pointers */
     C_Ptr->Next_Prod = NULL;
     C_Ptr->Next = NULL;
     C_Ptr->Num_Prods = 0;
     return ((Clau_Ptr) C_Ptr);
} /* End Create_Clause_Node */
```

```
void HOST_Get_Input_Data(infile)
char *infile;

/* This procedure simulates the functions of the
   Host computer:
   (1) Gets the input data from the given file, into
       an array, one product at a time. Creates a new
       product node, passes the number of literals in
       this product, the array pointer, and the product
       _node pointer to an encoding routine.
   (2) Links them as linked list of clauses. Each clause
       is a header of a linked-list of product nodes.
 */
{
   int i, Num_Lit;
   int Num_Clauses;
   Prod_Ptr Last_P;
   Clau_Ptr Last_C;
   void Product_Encoder();
   Clau_Ptr Create_Clause_Node();
   Prod_Ptr Create_Product_Node();
   void Release_Product_Node();
   void Echo_a_Clause();

   if ((fp = fopen(infile,"r"))==(FILE *) NULL) {
      printf("\n SORRY!..'%12s' NOT FOUND!\n",infile);
      exit(99);
      }
   else { /* Read the input data items */
      Num_Clauses = 0;
      for (i=0; i<MAX_LITS; i++)
         Literal_Arr[i] = PROD_DELIM; /* Init InBuf */
      Num_Lit = 0;
      HOST.Rcvr_Ptr = Create_Clause_Node();
      HOST.SOP_Ptr = Create_Product_Node();

      while (fscanf(fp,"%d",&Input_Literal)!= EOF){
         if (Input_Literal == CLAU_DELIM) {
            if (HOST.Host_Ptr == NULL)  {
            /* first clause */
               HOST.Host_Ptr = HOST.Rcvr_Ptr;
               Last_C = HOST.Rcvr_Ptr;
               }
            else if
                 (HOST.Rcvr_Ptr->Next_Prod != NULL) {

           /* Link successor clauses with the HOST
              Clause Header. Create a new Receiver
              clause node. Initialize relevant
              pointers. Update Clause counter.
            */
```

```
                Last_C->Next = HOST.Rcvr_Ptr;
                Last_C = Last_C->Next;
                } /* Product nodes exist in new clause */
             Num_Lit = 0;
             Last_P = NULL;
             HOST.Num_Clau_Possess++;
             HOST.Rcvr_Ptr = Create_Clause_Node();
             }/* Finish a clause */
          else if ((Input_Literal == PROD_DELIM) &&
                   (Num_Lit != 0)) {
             Product_Encoder(Num_Lit, Literal_Arr,
                             HOST.SOP_Ptr->Prod_Patt);
             /* must link it with the clause node
             */
             if (HOST.Rcvr_Ptr->Next_Prod == NULL) {
                HOST.Rcvr_Ptr->Next_Prod = HOST.SOP_Ptr;
                Last_P = HOST.SOP_Ptr;
                } /* 1st product */
             else {
                Last_P->Next = HOST.SOP_Ptr;
                Last_P = Last_P->Next;
                }
             Num_Lit = 0;
             HOST.Rcvr_Ptr->Num_Prods++;
             HOST.SOP_Ptr = Create_Product_Node();
             } /* Finish a product */

          /* Inputing valid literals.  Store them in the
             Literal_Arr storage. */
          else if ((abs(Input_Literal) <= MAX_LITS) &&
                   (Input_Literal!=0)) {
             Literal_Arr[Num_Lit++] = Input_Literal;
             } /* Literals in the range of +/- 124 */
          }/* while not eof */

          free ((Clau_Ptr) HOST.Rcvr_Ptr);
          HOST.Rcvr_Ptr = NULL;
          Release_Product_Node(HOST.SOP_Ptr);
          HOST.SOP_Ptr = NULL;

          if (DEBUGS) {
              printf ("\nNUM OF INPUT CLAUSES = %4d",
                               HOST.Num_Clau_Possess);
              printf ("\nNULL = %#X", NULL);
              printf ("\nMAP_PROD = %#X", MAP_PROD);
              printf ("\nNIL_PROD = %#X", NIL_PROD);
              printf ("\nMAX_PROD = %#X", MAX_PROD);
              Print_Line('\n','#',74);
              }
       } /* Datafile exists */
    fclose(fp);
} /* End HOST_Get Input Data */
```

```
void HOST_Update_Mail_Status()

/* The HOST updates its status: if the number of
   clauses remains in the host memory is lower than
   the number of leaf nodes, the host will set its
   mail_box status to CLEAR.  This is done to
   prevent the root node to transfer data back to
   it.  The data in each BPP will be processed, and
   transfered from the upper levels downward the
   root.  The memory capacity of the BPP will prevent
   overloading a node.  As the last pair of clauses
   is multiplied at the root. The final products will
   be returned to the Host in a SOP expression.
   NOTE that the Host will monitor the leafnode_BPPs,
   and set their mail_boxes to FINISH when it has no
   more clauses to send them. */
{
   if (HOST.Mail_Box == CLEAR) {
      /* Stop communication until last SOP
      */
      if (HOST.Num_Clau_Possess <= 0)
         HOST.Mail_Box = FINISH;
      /* Otherwise, continue request root BPP to send more
         clauses back to fetch the leaf-node BPPs.
      */
      else if (GPFS[0].PMU->Num_Clau_Possess!=0)
         HOST.Mail_Box = 0;
      }
   /* else: Mail_Box is COMMUN. Leave it alone 'til
      it is CLEARed */

}/* HOST_Update_Mail_Status */



void HOST_Setup_Transfer_Header (Device_ID)
int Device_ID;
{
   int i;
   void Echo_a_Clause();

   i = Device_ID - LEAF_INX;

   /* Give it a new clause: Setup a transfer pointer
   */
   if (HOST.Leaf_Xfer[i] == NULL) {
      if (HOST.Host_Ptr != NULL) {
         HOST.Leaf_Xfer[i] = HOST.Host_Ptr;
         HOST.Host_Ptr = HOST.Host_Ptr->Next;
         HOST.Num_Clau_Possess--;
```

```
            HOST.Leaf_Xfer[i]->Next = NULL;
            GPFS[Device_ID].Mail_Box = COMMUN;
            }
         /* No more data is available from HOST.
         */
         else if ((HOST.Rcvr_Ptr == NULL) ||
            (HOST.Rcvr_Ptr->Next_Prod == NULL)) {
            GPFS[Device_ID].Mail_Box = FINISH;
            if DEBUGS printf("\n---- HOST: NO RESOURCES----");
            } /* Son! don't ask anymore. */
         /* else: ignore request, waiting for next pass. */
         }
      else GPFS[Device_ID].Mail_Box = COMMUN;

} /* End HOST_Setup_Transfer_Header */



void HOST_Fetch_Leaf_Node_BPPs (Device_ID)
int Device_ID;
{
   int i;
   Prod_Ptr Pp;
   void Transfer_a_Product_Node();

   i = Device_ID - LEAF_INX;
   /* Continue sending the clause previously allocated
   */
   if (HOST.Leaf_Xfer[i] != NULL) {
      Pp = HOST.Leaf_Xfer[i]->Next_Prod;

      if (Pp != NULL) {
         HOST.Leaf_Xfer[i]->Next_Prod = Pp->Next;
         Transfer_a_Product_Node (Device_ID, Pp);
         HOST.Leaf_Xfer[i]->Num_Prods--;
         GPFS[Device_ID].Mail_Box = COMMUN;
         }
      /* Empty clause left-over after sent all products
      */
      else { /* EOL */
         HOST.Leaf_Xfer[i]->Next_Prod = NULL;
         Transfer_a_Product_Node (Device_ID, MAX_PROD);
         free((Clau_Ptr) HOST.Leaf_Xfer[i]);
         HOST.Leaf_Xfer[i] = NULL;
         GPFS[Device_ID].Mail_Box = CLEAR;
         } /* Last product is sent. Clear request */
      }
} /* End HOST_Fetch_Leaf_Node_BPPs */
```

```
void HOST_Leaf_BPPs_Interface()
{
   int i, Device_ID;
   void HOST_Update_Mail_Status();
   void HOST_Setup_Transfer_Header();
   void HOST_Fetch_Leaf_Node_BPPs();

   HOST_Update_Mail_Status();

   /* Check if a clause is being sent to a leaf-node BPP.
      Otherwise, if request pending, and data available
      locate a clause to the requester: leave it in the
      pointer array. For example, with a three level DFT,
      indexes 0 to 3: four leaves BPPs, Device_IDs: 3,4,5,6:
      Leaf-node BPPs.  Last_Leaf should be in the range of
      [Leaf_Inx, Host_Inx).
   */
   for (i=0; i<NUM_PORT; i++) {
      Device_ID = i + 1 + LAST_LEAF;
      while (Device_ID >= Host_ID)
         Device_ID = (Device_ID-Host_ID) + LEAF_INX;

      /*  HOST!  PLEASE SEND ME AGAIN !
      */
      if (GPFS[Device_ID].Mail_Box == Host_ID)
         HOST_Setup_Transfer_Header(Device_ID);

      /* HOST: Communicating with leaf-nodes.
      */
      if (GPFS[Device_ID].Mail_Box == COMMUN)
         HOST_Fetch_Leaf_Node_BPPs (Device_ID);
      } /* for i */
   /* Last node being served
   */
   LAST_LEAF = (Device_ID<Host_ID) ? Device_ID
                                   : LEAF_INX;
} /* End HOST_Leaf_BPPs_Interface */




void HOST_Receive_Root_BPP_Products(New_Prod)
Prod_Ptr New_Prod;
{
   Clau_Ptr Create_Clause_Node();

   /* Last node in the transferred clause: Link the
      clause to Host's Header. Increment the clause
      counter.  Initialize all relevant parameters.
   */
```

```
    if ((HOST.Rcvr_Ptr == NULL) && (New_Prod != NULL) &&
        (New_Prod != MAX_PROD) && (New_Prod != NIL_PROD)) {
       HOST.Rcvr_Ptr = Create_Clause_Node();
       HOST.Rcvr_Ptr->Next_Prod = New_Prod;
       HOST.SOP_Ptr = New_Prod;
       HOST.Rcvr_Ptr->Num_Prods++;
       }
    else if (New_Prod != NULL) {
       if (New_Prod == MAX_PROD) { /* Clause Delimiter */
          if (HOST.Rcvr_Ptr->Next_Prod != NULL) {
             HOST.Rcvr_Ptr->Next = HOST.Host_Ptr;
             HOST.Host_Ptr = HOST.Rcvr_Ptr;
             if (++HOST.Num_Clau_Possess >= 2)
                 HOST.STATUS = ACTIVE;
             HOST.Num_Feedback_Clauses++;
             }
          else { /* Empty clause: eliminated */
             free((Clau_Ptr) HOST.Rcvr_Ptr);
             }
          HOST.Rcvr_Ptr = NULL;
          HOST.SOP_Ptr = NULL;
          }
       /* Link product with the existing Rcvr Clause
       */
       else if (New_Prod != NIL_PROD) {
          if (HOST.Rcvr_Ptr->Next_Prod == NULL)
          /* 1st node */
             HOST.Rcvr_Ptr->Next_Prod =
             HOST.SOP_Ptr = New_Prod;
          else {
          /* Second to SEL_PRODS nodes */
             HOST.SOP_Ptr->Next = New_Prod;
             HOST.SOP_Ptr = HOST.SOP_Ptr->Next;
             }
          HOST.Rcvr_Ptr->Num_Prods++;
          }
       }
} /* HOST_Receive_Root_BPP_Products */



void Init_HOST(infile)
char *infile;
{
    int i;
    void HOST_Get_Input_Data();

    HOST.STATUS = ACTIVE;
    HOST.Mail_Box = CLEAR;
```

```
   HOST.Total_Exec_Cycles = 0;
      /* Total number of execution steps */

   HOST.Num_Feedback_Clauses = 0;
   HOST.Num_Clau_Possess = 0;
      /* Number of clauses available for BPPs */

   HOST.Num_Final_Prods = 0;
      /* Num of products in the final SOP */

   HOST.Host_Ptr = NULL;
      /* Given clauses in the original GPF */

   HOST.Rcvr_Ptr = NULL;
      /* Clauses returned from the root BPP */

   HOST.SOP_Ptr = NULL;
      /* Pointer to the final Result products */

   for (i=0; i<NUM_LEAF; i++) HOST.Leaf_Xfer[i] = NULL;
      /* Ptrs to clauses sent to Leaf-node */

   Normal_Termination = TRUE;
   HOST_Get_Input_Data(infile);
   printf("\nINITIALIZED HOST STATUS");
} /* Init_HOST */



Boolean GPFS_Process_Terminated()
{
   int i;
   Boolean Stop;

   Stop = FALSE;
   if (HOST.STATUS == IDLE) {
      if (Normal_Termination == FALSE) Stop = TRUE;
      else for (i=1; i<NUM_BPPS; i++)
          if (GPFS[i].PMU->Clause_Header==NULL) Stop = TRUE;
      }
   else if (HOST.Num_Final_Prods >= SEL_PRODS) {
       for (i=0; i<NUM_BPPS; i++)
          if ((GPFS[i].PMU->Num_Clau_Possess==0)
             && (GPFS[i].Mail_Box==FINISH)) Stop = TRUE;
       }
   return(Stop);
} /* End GPFS Process Terminated */
```

```
void main(a1,a2)
int a1;
char *a2[];
{
   int i, Device_ID;
   Boolean j;
   void Init_HOST();
   void Create_GPFS();
   void GPFS_Report();
   void BPP_Operations();
   void GPFS_Final_Report();
   void HOST_Leaf_BPPs_Interface();
   Boolean GPFS_Process_Terminated();

   /* INVOCATION:
    > HOST  In_File Num_DFTs Sapa_Siz Ports   Debug
      *a2[0] *a2[1] *a2[2]   *a2[3]  *a2[4]  *a2[5]
      e.g.,  Try.dat   3      128      4    0=N,1=Y
      Notes:  Data_filename must be shorter than 25
      chars. Debug option must be either '0' or '1'.
      Num_DFTs, Sapa_Size, and Number of Host Ports
      must be greater than zero. The number of leaf
      BPPs that the Host can serve per cycle (Ports)
      is assumed to be equal or less than the total
      leaf-BPPs.
   */
   Report_Frequency = 1; /* Default */
   NUM_DFTS = (int) atoi(a2[2]);
   if (NUM_DFTS > MAX_NUM_DFTS) NUM_DFTS = MAX_NUM_DFTS;
   SAPA_SIZ = (int) atoi(a2[3]);
   if (SAPA_SIZ < MIN_SAPA_SIZ) SAPA_SIZ = MIN_SAPA_SIZ;
   Create_GPFS();

   NUM_PORT = (int) atoi(a2[4]);
   if ((NUM_PORT <= 0) || (NUM_PORT > NUM_LEAF))
        NUM_PORT = NUM_LEAF;

   LAST_LEAF = LEAF_INX-1;
   printf("\nDFT =%4d, SAPA =%4d, PORT =%4d",
            NUM_DFTS, SAPA_SIZ, NUM_PORT);
   if (*a2[5] == '1') { /* Debug Option = ON */
      printf("\nEnter Report Frequency: ");
      scanf ("%d",&Report_Frequency);
      if ((Report_Frequency <= 0) ||
          (Report_Frequency > 1000))
         Report_Frequency = 50;
      printf("%d", Report_Frequency);
      DEBUGS = TRUE;
      }
   Init_HOST(a2[1]);
   Print_Legends();
   GPFS_Report();
```

```
   start_time = get_time();

   while (GPFS_Process_Terminated() == FALSE) {
      /* Switching execution b/w BPPs at every step.
      */
      for (Device_ID = 0;
           Device_ID < NUM_BPPS; Device_ID++) {
         if (DEBUGS) {
            if ((HOST.Total_Exec_Cycles %
                   Report_Frequency) == 0)
            printf("\nBPP DEVICE#%2d:",Device_ID);
            }
         BPP_Operations(Device_ID);
         }
      HOST_Leaf_BPPs_Interface();
      HOST.Total_Exec_Cycles++;
      if ((DEBUGS) && ((HOST.Total_Exec_Cycles %
                        Report_Frequency) == 0)) {
         GPFS_Report();
         printf("\n.............................");
         printf("........Press any key to continue");
         scanf("%c",&i);
         }
      } /* while */

   GPFS_Final_Report();
   finish_time = get_time();
   printf("\nElapsed Time = %lf seconds.\n",
                  how_long(start_time, finish_time));
} /* End PROGRAM GPFS */
```

```
/*-------------------------------------------------*/
/* GENERALIZED PROPOSITIONAL FORMULA SOLVER - Part#2 */
/*-------------------------------------------------*/
/* LINK LIST VERSION.                              */
/*   This program simulates the operations of the  */
/*   Control Unit in a PMU.                        */
/*-------------------------------------------------*/
/*     Filename:       GPF.C                       */
/*     Created:        FRI  02 JUN 1989      20:00 */
/*     Last Edited:    FRI  12 OCT 1989      23:45 */
/*-------------------------------------------------*/


void Create_BPP_Node(Device_ID)
int Device_ID;
{
   Pmu_Ptr Create_PMU();
   Spru_Ptr Create_SPRU();

   GPFS[Device_ID].Status = ACTIVE; /* Default */
   GPFS[Device_ID].Num_Idles = 0;
   GPFS[Device_ID].Num_Clauses_Received = 0;
   GPFS[Device_ID].SEL_Sapa = FALSE;

   /* Calculate the indexes of the predecessor BPP
      nodes. The root node is 0, counting upward the
      tree hierarchy: for three levels DFT, 7 nodes
      totally. The Host is assumed to be Device_ID#
      NUM_BPPS = 7.
   */
   if (Device_ID < LEAF_INX) {
      GPFS[Device_ID].Left_ID = Device_ID*2 +1;
      GPFS[Device_ID].Rite_ID =
                  GPFS[Device_ID].Left_ID + 1;
      }
   else /* Leaf_Nodes: Communicates to HOST */
      GPFS[Device_ID].Left_ID =
      GPFS[Device_ID].Rite_ID = Host_ID;

   /* Calculate the indexes of the successor BPP nodes.
   */
   if (Device_ID==0) GPFS[Device_ID].Succ_ID = Host_ID;
   else GPFS[Device_ID].Succ_ID = (Device_ID-1)/2;
   GPFS[Device_ID].MAX_CLAU = MAX_CLAUS;
   GPFS[Device_ID].Mail_Box =
       GPFS[Device_ID].Left_ID; /* Left: SENDME */
   GPFS[Device_ID].Xmit_Ptr = NULL;
   GPFS[Device_ID].PMU  = Create_PMU ();
   GPFS[Device_ID].SPRU = Create_SPRU();
} /* End Create BPP Node */
```

```
void Create_GPFS()
{
   int i, j;
   void Create_BPP_Node();
   Prod_Ptr Create_Product_Node();

   NUM_BPPS = j = 1; /* Number of BPPs in the GPFS */
   for (i=1; i<NUM_DFTS; i++) {
      j *= 2;
      NUM_BPPS += j;
      }
   Host_ID  = NUM_BPPS;
   NUM_LEAF = j;
   LEAF_INX = NUM_BPPS - NUM_LEAF;
   SEL_PRODS = (int) sqrt((double) (SAPA_SIZ * 2));
       /* Sapa_Siz is the number of PEs in a SAPA.
          SEL_PRODS is the number of Cartesian products
          selected from the SAPA outputs to be delivered
          to the BPP at the next DFT level.
       */
   if (DEBUGS == TRUE) {
      printf("\nNum_BPPs = %3d, NumLeaf = %3d",
                NUM_BPPS, NUM_LEAF);
      printf("\nLeaf_Inx = %3d, MaxProds = %3d",
                LEAF_INX, SEL_PRODS);
       /* Number of products in a sorted clause */
      }
   for (i=0; i<NUM_BPPS; i++)  Create_BPP_Node(i);
   MAP_PROD = Create_Product_Node();
   MAX_PROD = Create_Product_Node();
   NIL_PROD = Create_Product_Node();
   printf("\nSUCCESSFULLY CREATED THE GPFS");
} /* End Create GPFS */



void Accept_and_Link_SAPA_Results (Device_ID, Sort_Prod)
int Device_ID;
Prod_Ptr Sort_Prod;
/*
   Link_Untagged_Products: Check and reject
   dominating products which are tagged
   by the SPRU/SAPAs. Link the resultant product
   nodes to Clau_Link otherwise. */
{
   Clau_Ptr Create_Clause_Node();
   void Echo_a_Clause();

   Sort_Prod->Next = NULL;
```

```
/* A valid product is received. Check if Clau node does not
   exist.  Create the header, link the new product in.
*/
if (GPFS[Device_ID].PMU->Clau_Link == NULL) {
   if ((Sort_Prod != MAX_PROD) &&
       (Sort_Prod != NIL_PROD)) {
       GPFS[Device_ID].PMU->Clau_Link =
            Create_Clause_Node();
       GPFS[Device_ID].PMU->Clau_Link->Next_Prod=Sort_Prod;
       GPFS[Device_ID].PMU->Prod_Link = Sort_Prod;
       GPFS[Device_ID].PMU->Clau_Link->Num_Prods++;
       }
    }
else if (Sort_Prod != NULL) {
   if (Sort_Prod == MAX_PROD) {
   /* End of Clause.  Link Clau_link to the header.
      Init other pointer parameters.
      If the resultant clause is empty: remove it.
   */
      if (DEBUGS == TRUE) {
         printf("\n\tLink a clause from SAPA:");
         Echo_a_Clause (GPFS[Device_ID].PMU->Clau_Link);
         }
      if(GPFS[Device_ID].PMU->Clau_Link->Next_Prod!=NULL){
         GPFS[Device_ID].PMU->Clau_Link->Next =
             GPFS[Device_ID].PMU->Clause_Header;
         GPFS[Device_ID].PMU->Clause_Header =
             GPFS[Device_ID].PMU->Clau_Link;
         GPFS[Device_ID].Num_Clauses_Received++;
         GPFS[Device_ID].PMU->Num_Clau_Possess++;
         } /* Product nodes exist in the new clause */

      /*   Empty clause resulted: no solution
      */
      else {
         printf("\n### Device#%2d: ", Device_ID);
         printf("SAPA RETURNED EMPTY CLAUSE ###");
         printf("\n            NO SOLUTION FOUND");
         Normal_Termination = FALSE;
         HOST.STATUS = IDLE;
         HOST.Mail_Box = FINISH;
         free ((Clau_Ptr) GPFS[Device_ID].PMU->Clau_Link);
         }
      GPFS[Device_ID].PMU->Clau_Link = NULL;
      GPFS[Device_ID].PMU->Prod_Link = NULL;
      }/* Finish a clause: MAX_PROD is the delimiter */


   /* Not a Dominating product returned from a SAPA, and
      the new product is neither a clause nor delimiter.
   */
   else if (Sort_Prod != NIL_PROD) {
```

```
         /* The given product is the first node in the link
         */
         if(GPFS[Device_ID].PMU->Clau_Link->Next_Prod==NULL){
            GPFS[Device_ID].PMU->Clau_Link->Next_Prod =
            GPFS[Device_ID].PMU->Prod_Link = Sort_Prod;
            }
         else { /* Succeeding nodes */
            GPFS[Device_ID].PMU->Prod_Link->Next = Sort_Prod;
            GPFS[Device_ID].PMU->Prod_Link =
                        GPFS[Device_ID].PMU->Prod_Link->Next;
            }
         GPFS[Device_ID].PMU->Clau_Link->Num_Prods++;
         } /* Not a dominating product */
      } /* Clau_Link was already setup */
}  /* End Accept_and_Link_SAPA_Results */



Boolean BPP_Empty_Predecessor (Device_ID)
int Device_ID;
{
   int Left_ID, Rite_ID;

   if (Device_ID<LEAF_INX) {
      Left_ID = GPFS[Device_ID].Left_ID;
      Rite_ID = GPFS[Device_ID].Rite_ID;

      if ((GPFS[Left_ID].PMU->Clause_Header == NULL) &&
          (GPFS[Left_ID].PMU->Clau_Link == NULL) &&
          (GPFS[Left_ID].PMU->Rcvr_Ptr1 == NULL) &&
          (GPFS[Left_ID].PMU->CPG_Status== DONE) &&
          (GPFS[Left_ID].Xmit_Ptr  == NULL) &&
          (GPFS[Left_ID].SPRU->SAPAL->EMPTY == TRUE) &&
          (GPFS[Left_ID].SPRU->SAPAR->EMPTY == TRUE) &&

          (GPFS[Rite_ID].PMU->Clause_Header == NULL) &&
          (GPFS[Rite_ID].PMU->Clau_Link == NULL) &&
          (GPFS[Rite_ID].PMU->Rcvr_Ptr1 == NULL) &&
          (GPFS[Rite_ID].PMU->CPG_Status== DONE) &&
          (GPFS[Rite_ID].Xmit_Ptr  == NULL) &&
          (GPFS[Rite_ID].SPRU->SAPAL->EMPTY == TRUE) &&
          (GPFS[Rite_ID].SPRU->SAPAR->EMPTY == TRUE))
      return(TRUE);
      else return(FALSE);
      }
   else { /* Leaf Nodes */
      if ((HOST.Host_Ptr==NULL) && (HOST.Rcvr_Ptr==NULL))
      return(TRUE);
      else return(FALSE);
      }
} /* End BPP_Empty_Predecessors */
```

```
void BPP_Support_Successors(Device_ID)
int Device_ID;

/* Check if resource is available. Transfer clauses by
   handling pointers accross devices (BPPS). Clear request
   (Mail_Box) after responsed to the child node.  This
   procedure is applied to the nodes not in the leaf level of
   the DFT: Clauses transferred between parent BPPs to child
   BPPs, one product node at a time. */
{
   Pmu_Ptr  Up;
   Prod_Ptr Pp;
   int Succ_ID, Left_ID, Rite_ID;
   Clau_Ptr Create_Clause_Node();
   void HOST_Receive_Root_BPP_Products();
   void Transfer_a_Product_Node();
   Boolean BPP_Empty_Predecessor();

   Up = GPFS[Device_ID].PMU; /* Parent */
   Succ_ID = GPFS[Device_ID].Succ_ID;
   Left_ID = GPFS[Device_ID].Left_ID;
   Rite_ID = GPFS[Device_ID].Rite_ID;


  /*------ ROOT BPP - HOST INTERFACE ----------------*/

   if (Device_ID == 0) {
      if (HOST.Mail_Box == Device_ID) {
         /* Product nodes exist in the new clause */

         if (GPFS[Device_ID].Xmit_Ptr == NULL) {
            if (Up->Clause_Header != NULL) {
               if ((Up->Num_Clau_Possess >= 2) ||
                  ((Up->Num_Clau_Possess >= 1) &&
                  (BPP_Empty_Predecessor(Device_ID)==TRUE))){
                 GPFS[Device_ID].Xmit_Ptr= Up->Clause_Header;
                 Up->Clause_Header = Up->Clause_Header->Next;
                  GPFS[Device_ID].Xmit_Ptr->Next = NULL;
                  Up->Num_Clau_Transfer++;
                  Up->Num_Clau_Possess--;
                  HOST.Mail_Box = COMMUN;
                  if (DEBUGS==TRUE) printf
                           ("\nRoot returns a clause to HOST");
                  } /* move a clause from Header */
               } /* PMU still contains some more clauses */
            else if (HOST.Num_Clau_Possess<=0) /*NUM_LEAF*/
               HOST.Mail_Box = FINISH;
            } /* Preparing for transmission */
         else if (GPFS[Device_ID].Xmit_Ptr->Next_Prod != NULL)
            HOST.Mail_Box = COMMUN;
         } /* Current device is requested to send data */
```

```
/* Pointer is already set. Transmit another product
*/
if (HOST.Mail_Box == COMMUN) {/* Xfer in progress */
   if (GPFS[Device_ID].Xmit_Ptr != NULL) {
      Pp = GPFS[Device_ID].Xmit_Ptr->Next_Prod;
      if (Pp != NULL) {
         GPFS[Device_ID].Xmit_Ptr->Next_Prod =
                                        Pp->Next;
         Pp->Next = NULL;
         }
      /* Completely transferred the given clause
         to the successor
      */
      else {
         Pp = MAX_PROD; /* Delimiter */
         free( (Clau_Ptr) GPFS[Device_ID].Xmit_Ptr);
         GPFS[Device_ID].Xmit_Ptr = NULL;
         HOST.Mail_Box = CLEAR;
         }
      HOST_Receive_Root_BPP_Products(Pp);
      }
   else HOST.Mail_Box = CLEAR;
   }
/* Final SOP product. Transferred back to the Host by
   handling Clause Pointer. Increment Exec_Steps by
   number of products in the transmitted clause.
*/
else if (HOST.Mail_Box == FINISH) {
   if ((GPFS[Device_ID].Xmit_Ptr == NULL) &&
       (BPP_Empty_Predecessor(Device_ID) == TRUE) &&
       (Up->Clau_Link == NULL) &&
       (Up->Rcvr_Ptr1 == NULL) &&
       (Up->CPG_Status== DONE) &&
       (GPFS[Device_ID].SPRU->SAPAL->EMPTY == TRUE) &&
       (GPFS[Device_ID].SPRU->SAPAR->EMPTY == TRUE) &&
       ((Up->Clause_Header == NULL) ||
        ((Up->Clause_Header != NULL) &&
         (Up->Num_Clau_Possess == 1)))) {
      HOST.Rcvr_Ptr = Up->Clause_Header;
      HOST.Num_Final_Prods=Up->Clause_Header->Num_Prods;
      HOST.SOP_Ptr = HOST.Rcvr_Ptr->Next_Prod;
      Up->Clause_Header = NULL;
      HOST.STATUS = IDLE;
      }
   }
}/* Root BPP communicates with Host */
```

```
/*------------- PRED BPPS - SUCC BPPS INTERFACE ----------*/

 else { /* BPPs at upper levels */
    if (GPFS[Succ_ID].Mail_Box == Device_ID) {
    /* Request to send */

       if (GPFS[Device_ID].Xmit_Ptr == NULL) {
          if (Up->Clause_Header != NULL) {
             if ((Up->Num_Clau_Possess >= 2) ||
                ((Up->Num_Clau_Possess >= 1) &&
                (BPP_Empty_Predecessor(Device_ID)==TRUE))){

                if (DEBUGS==TRUE)
                  printf("\n\tTransfer a clause to BPP#%2d",
                         Succ_ID);
                GPFS[Device_ID].Xmit_Ptr=Up->Clause_Header;
                Up->Clause_Header=Up->Clause_Header->Next;
                GPFS[Device_ID].Xmit_Ptr->Next = NULL;
                Up->Num_Clau_Possess--;
                Up->Num_Clau_Transfer++;
                GPFS[Succ_ID].Mail_Box =
                (GPFS[Succ_ID].Left_ID==Device_ID) ? LCOMM
                                                  :RCOMM;
                } /* More data is available */
             } /* Move a clause from Header */

          else if (BPP_Empty_Predecessor(Device_ID)==TRUE)
             GPFS[Succ_ID].Mail_Box = FINISH;
          } /* Preparing for transmission */

       else if(GPFS[Device_ID].Xmit_Ptr->Next_Prod!=NULL){
          GPFS[Succ_ID].Mail_Box =
          (GPFS[Succ_ID].Left_ID==Device_ID) ? LCOMM
                                             : RCOMM;

          }
       else {
          free ((Clau_Ptr) GPFS[Device_ID].Xmit_Ptr);
          GPFS[Device_ID].Xmit_Ptr = NULL;
          GPFS[Succ_ID].Mail_Box = CLEAR;
          }
       } /* DEVICE_ID: Current device is requested to send
                                                    data */
    /* Pointer is already set. Transmit another product
    */
    if (((GPFS[Succ_ID].Mail_Box == LCOMM) &&
         (GPFS[Succ_ID].Left_ID == Device_ID)) ||
        ((GPFS[Succ_ID].Mail_Box == RCOMM) &&
         (GPFS[Succ_ID].Rite_ID == Device_ID))) {
       if (GPFS[Device_ID].Xmit_Ptr != NULL) {
          Pp = GPFS[Device_ID].Xmit_Ptr->Next_Prod;
          if (Pp != NULL) {
             GPFS[Device_ID].Xmit_Ptr->Next_Prod=Pp->Next;
```

```
                Transfer_a_Product_Node(Succ_ID, Pp);
                GPFS[Device_ID].Xmit_Ptr->Num_Prods--;
                GPFS[Succ_ID].Mail_Box =
                (GPFS[Succ_ID].Left_ID == Device_ID) ? LCOMM
                                                    : RCOMM;
                }
             else {
                Transfer_a_Product_Node(Succ_ID, MAX_PROD);
                free( (Clau_Ptr) GPFS[Device_ID].Xmit_Ptr);
                GPFS[Device_ID].Xmit_Ptr = NULL;
                GPFS[Succ_ID].Mail_Box = CLEAR;
                }
             }
          else GPFS[Succ_ID].Mail_Box = CLEAR;
          } /* COMMUN: Transfering data */
       } /* BPP nodes above the root */
} /* BPP_Support_Successors */



void BPP_Interface_with_Predecessors(Device_ID)
int Device_ID;

/* Check current status. If the number of clauses
   are smaller than the maximum memory capacity,
   ask parent nodes for more data. Check the parents'
   resources prior to placing the request.  The mail
   should indicate LEFT or RIGHT device to response.
   If both predecessors have equal/or no more clauses
   available, place the request to the Left node. Also,
   this part must check if the current device is a
   leaf-node: request the HOST directly. */
{
   int Left_ID, Rite_ID;

   if ((GPFS[Device_ID].Status == IDLE) ||
       (GPFS[Device_ID].PMU->Num_Clau_Possess <
        GPFS[Device_ID].MAX_CLAU)) {

      if (Device_ID < LEAF_INX) {
          Left_ID = GPFS[Device_ID].Left_ID;
          Rite_ID = GPFS[Device_ID].Rite_ID;
          if ((GPFS[Rite_ID].PMU->Num_Clau_Possess!=0) ||
              (GPFS[Left_ID].PMU->Num_Clau_Possess!=0)) {

             /* Current Mail_Box is CLEAR. Request to send
                (if neccessary)
             */
             if ((GPFS[Device_ID].Mail_Box == CLEAR) ||
                 (GPFS[Device_ID].Mail_Box == FINISH)) {
```

```
                if (GPFS[Rite_ID].PMU->Num_Clau_Possess>
                    GPFS[Left_ID].PMU->Num_Clau_Possess)
                    GPFS[Device_ID].Mail_Box = Rite_ID;
                else GPFS[Device_ID].Mail_Box = Left_ID;
                }
             /* else:  Mail_Box is COMMUN, or FINISH,
                or Device_ID: Requests has been placed.
                Waiting for services. Services might not
                be available.
              */
             } /* Parent nodes still have data to transfer */
       }
      else { /* Leaf nodes: Request data from HOST */
         if ((GPFS[Device_ID].Mail_Box == CLEAR)  ||
             ((GPFS[Device_ID].Mail_Box == FINISH) &&
              ((HOST.STATUS == ACTIVE) ||
               (HOST.Rcvr_Ptr != NULL) ||
               ((HOST.Host_Ptr != NULL) &&
                (HOST.Num_Clau_Possess > 0)) )))
           GPFS[Device_ID].Mail_Box = Host_ID;
         }
      }/* Condition for placing request. */
} /* BPP_Interface_with_Predecessors */



void BPP_Operations(Device_ID)
int Device_ID;
{
   Prod_Ptr Cart_Prod,
            Sort_Prod;
   Prod_Ptr PMU_Operations();
   Prod_Ptr SPRU_Operations();
   void BPP_Support_Successors();
   void Accept_and_Link_SAPA_Results();
   void BPP_Interface_with_Predecessors();

   /* Update BPP Operating Status:
      Remember that the header might contain no clause,
      but the CPG maybe BUSY generating more products,
      or the SPRU may pipeline a clause in its register
      storages.  */
   if ((GPFS[Device_ID].PMU->Num_Clau_Possess<=1) &&
       (GPFS[Device_ID].PMU->CPG_Status == DONE) &&
       (GPFS[Device_ID].SPRU->SAPAL->EMPTY) &&
       (GPFS[Device_ID].SPRU->SAPAR->EMPTY) &&
       (GPFS[Device_ID].PMU->Clau_Link == NULL)) {
        GPFS[Device_ID].Num_Idles++;
        GPFS[Device_ID].Status = IDLE;
       }
```

```
   else GPFS[Device_ID].Status = ACTIVE;
   BPP_Interface_with_Predecessors(Device_ID);

   /* Device is ACTIVE: Proceed BPP execution.
      This block must be IF block. Cannot be multiplexed
      with the previous block. Since, the requests may
      still be placed even if the device is not in IDLE
      state.  */
   if (GPFS[Device_ID].Status == ACTIVE) {
      Cart_Prod = PMU_Operations (Device_ID);
      Sort_Prod = SPRU_Operations(Device_ID, Cart_Prod);
      if (Sort_Prod != NULL) {
         Sort_Prod->Next = NULL;
         Accept_and_Link_SAPA_Results(Device_ID, Sort_Prod);
         }
      } /* ACTIVE MODE: NORMAL OPERATIONS */

   /* Check mail boxes. Respond to successor requests.
      If the successor waiting for more data, then direct
      the SAPA to transfer the next output clause to it.
      Otherwise, the new node will be accummulated in the
      current BPP.  Since SAPA provides the only path to
      the successor node, their requests might not be
      satisfied immediately, but must be synchronized to
      the start of a clause output from Left/or Right SAPA
      in the SPRU of the parents' BPP nodes.  */
   BPP_Support_Successors(Device_ID);
} /* End BPP Operations */
```

```
/*-------------------------------------------------*/
/* GENERALIZED PROPOSITIONAL FORMULA SOLVER - Part#3 */
/*-------------------------------------------------*/
/* LINK LIST VERSION.                              */
/* This program simulates the operation of the CTG  */
/*     in a PMU.                                   */
/*-------------------------------------------------*/
/*     Filename:       PMU.C                       */
/*     Created:        FRI  02 JUN 1989      20:00 */
/*     Last Edited:    FRI  13 OCT 1989      22:45 */
/*-------------------------------------------------*/


Pmu_Ptr Create_PMU()
{
   Pmu_Ptr P;
   Clau_Ptr Create_Clause_Node();

   if ((P = (Pmu_Ptr)
      malloc (sizeof(Pmu_Node))) == NULL) {
      printf("\nErrors in creating PMU nodes.");
      exit(11);
      }
   P->Num_Clau_Possess = 0;
   P->Num_Clau_Transfer= 0;
   P->Clause_Header = NULL;
   P->Rcvr_Ptr1 = NULL;
   P->Rcvr_Ptr2 = NULL; /* associated with Rcvr_Ptr1 */
   P->Clau_Link = NULL;

   P->Prod_Link = NULL; /* associated with Clau_Link */
   P->CPG_Ptr1  = P->CPG_Ptr2  = NULL;
   P->CPG_Ptr3  = P->CPG_Ptr4  = NULL;
   P->CPG_Status= DONE;
   P->CPG_Count = 0;
   return( (Pmu_Ptr) P);
} /* End Create_PMU */



Element Cost_Contradiction_Checker(Prod_P)
Prod_Ptr Prod_P;
/*
   Evaluate the cost of the given product,
   and check for contradiction.  The cost
   is written into bit locations 120-125.
   Return Zero if contradiction detected */
{
   int i, j;
   Element pattern;
   Element cost = 0;
```

```
    /* Check 16 Elements per a long word [Product Pattern]
       16 * 16 = 256 bit GPF word. The Cost field and Tag field
       are masked out first, to avoid being treated as
       contradictions. */

    *(Prod_P->Prod_Patt+(WORD_SIZ-1)) |= 0xFF00;

    for (i=0; i<WORD_SIZ; i++) {
      /* check 8 literals per 16 bit word [Element]
         2 * 8 = 16 bit word.
      */
       for (j=0; j<ELT_LEN; j++) {
          pattern= *(Prod_P->Prod_Patt+i)&CHK_MASK[j];
          if (pattern == 0) {
            /* Contradiction: Zero out the Tag and Cost fields
            */
            *(Prod_P->Prod_Patt+(WORD_SIZ-1))&=0x00FF;
             return(0);  /* contradiction */
             }
          else if (pattern != CHK_MASK[j]) {
             /* not a don't care: count the literals */
             if (i<WORD_SIZ-1) cost++;
             else if (j <= ELT_LEN/2) cost++;
             /* else: Tag & Cost field: do nothing */
             }
          /* else: don't care -- skip */
          }
       }
    /* Write 7 bit cost field. Set Tag bit (MSB) high.
    */
    cost = ((cost << 8) & 0xFF00) | 0x80FF;
    *(Prod_P->Prod_Patt+(WORD_SIZ-1)) &= cost;
    return(cost);
} /* End Cost_Contradiction_Checker */



void Transfer_a_Product_Node(Device_ID, New_Prod)
int Device_ID;
Prod_Ptr New_Prod;
/* A product node is sent from the predecessor
   (or Host) to the successor node. It must be
   linked to Clau_Ptr location.  The header of
   this clause is pointed to by Prod_Ptr.  Upon
   recepton of an empty product (NIL_PROD), the
   link is completed. The new clause is linked
   to the main PMU linked-list, headed by Clause
   Header.  The clause counter is incremented.
   Remember to increment the number of products
   in the current clause node. */
```

```
{
   Pmu_Ptr U_Ptr;
   Clau_Ptr Create_Clause_Node();
   void Echo_a_Clause();

   New_Prod->Next = NULL;
   U_Ptr = GPFS[Device_ID].PMU;

   if (U_Ptr->Rcvr_Ptr1 == NULL) {
      if ((New_Prod != NULL) && (New_Prod != MAX_PROD) &&
          (New_Prod != NIL_PROD)) {
          U_Ptr->Rcvr_Ptr1 = Create_Clause_Node();
          U_Ptr->Rcvr_Ptr1->Next_Prod = New_Prod;
          U_Ptr->Rcvr_Ptr2 = New_Prod;
          U_Ptr->Rcvr_Ptr1->Num_Prods++;
          }
       }
   /* Last node in the clause, Link the clause to the Header.
      Check number of clauses the PMU currently possesses.
      Update the counter.  Create new Rcvr node.
   */
   else if (New_Prod != NULL) {
      if (New_Prod == MAX_PROD) { /* EOL node */
         if (U_Ptr->Rcvr_Ptr1->Next_Prod != NULL) {
            U_Ptr->Rcvr_Ptr1->Next = U_Ptr->Clause_Header;
            U_Ptr->Clause_Header =  U_Ptr->Rcvr_Ptr1;
            U_Ptr->Num_Clau_Possess++;
            GPFS[Device_ID].Num_Clauses_Received++;
            if (DEBUGS==TRUE)
                printf("\n\tNew Clause Received:",
                      U_Ptr->Num_Clau_Possess);
            }
         else  /* Empty clause: eliminated */
            free((Clau_Ptr) U_Ptr->Rcvr_Ptr1);
         U_Ptr->Rcvr_Ptr1 = U_Ptr->Rcvr_Ptr2 = NULL;
         }
      else if (New_Prod != NIL_PROD) {
         if (U_Ptr->Rcvr_Ptr1->Next_Prod != NULL) {
            U_Ptr->Rcvr_Ptr2->Next = New_Prod;
            U_Ptr->Rcvr_Ptr2 = U_Ptr->Rcvr_Ptr2->Next;
            }
         else { /* first node */
            U_Ptr->Rcvr_Ptr1->Next_Prod = New_Prod;
            U_Ptr->Rcvr_Ptr2 = U_Ptr->Rcvr_Ptr1->Next_Prod;
            }
         U_Ptr->Rcvr_Ptr1->Num_Prods++;
         }
      }
} /* End Transfer a Product Node */
```

```
Prod_Ptr Generate_a_Cartesian_Prod (P1_Ptr, P2_Ptr)
Prod_Ptr P1_Ptr,
         P2_Ptr;
/* Multiply two given product nodes. Return the
   address of the newly created node. Its pattern
   is the result of logically ANDing the patterns
   of the 2 given nodes. The returned product node
   will be sent to the SPRU (SAPA). */
{
   int i;
   Element Prod_Cost;
   Prod_Ptr New_Prod_P;
   Prod_Ptr Create_Product_Node();
   Element Cost_Contradiction_Checker();
   void Release_Product_Node();
   New_Prod_P = Create_Product_Node();
   for (i=0; i<WORD_SIZ; i++) {
      *(New_Prod_P->Prod_Patt+i) =
          *(P1_Ptr->Prod_Patt+i) & *(P2_Ptr->Prod_Patt+i);
      }
   /* Compute the product cost, and check for contradiction.
      Clear the Tag bit if any contradiction is detected.
   */
   if ((Prod_Cost = Cost_Contradiction_Checker
       (New_Prod_P)) != 0) return(New_Prod_P);
   else { /* Contradictory product detected.
             Remove the new node */
      if (DEBUGS==TRUE) {
          printf("\n\tContradiction.");
      /** printf("\n\tProd#1 =");
          Echo_a_Product(P1_Ptr);
          printf("\n\tProd#2=");
          Echo_a_Product(P2_Ptr); **/
          }
      Release_Product_Node(New_Prod_P);
      return(NIL_PROD);
      }
} /* End Generate_a_Cartesian_Prod */



Prod_Ptr CPG_Operations (Device_ID)
int Device_ID;
/*
   Multiply out two given clauses. Release both
   clauses after accomplished the task. Multiply
   once, return a new product at every invocation.
   Results could be discarded by PMU if including
   contradiction.  Reset pointers when DONE.
   Pointers to the 2 given Clauses: CPG_Ptr1, CPG_Ptr2
   Pointers to current products:    CPG_Ptr3, CPG_Ptr4
*/
```

```
{
   Pmu_Ptr PMU_P;
   Prod_Ptr Send_Prod;
   void Release_a_Clause();
   void GPFS_Report();
   void Echo_a_Clause();
   Prod_Ptr Generate_a_Cartesian_Prod();

   PMU_P = GPFS[Device_ID].PMU;
   if ((PMU_P->CPG_Status==BUSY) &&
       (PMU_P->CPG_Ptr1 != NULL) &&
       (PMU_P->CPG_Ptr2 != NULL)) {
      /* If Send_Prod is a dominating product: its tag bits
         are cleared.  Actual returned pointer is NIL_PROD.
      */
      if ((Send_Prod=Generate_a_Cartesian_Prod
           (PMU_P->CPG_Ptr3,PMU_P->CPG_Ptr4))!=NIL_PROD)
         PMU_P->CPG_Count++;

      /* CHECK & UPDATE PROCESSING POINTERS:
         Ptr3 traverses faster. Reset Ptr3, advance
         Ptr4. As Ptr4 approaches the end of list
         (NULL), the CPG has completely multiplied two
         given clauses together.
      */
      if (PMU_P->CPG_Ptr3->Next != NULL) {
         PMU_P->CPG_Ptr3 = PMU_P->CPG_Ptr3->Next;
         }
      else { /* Ptr3 approaches the end of list */
         if (PMU_P->CPG_Ptr4->Next != NULL) {
            PMU_P->CPG_Ptr4 = PMU_P->CPG_Ptr4->Next;
            PMU_P->CPG_Ptr3 = PMU_P->CPG_Ptr1->Next_Prod;
            }
         else {
         /* Completely multiplied two given clauses */
         /* Release the clauses. Reset the pointers */
             PMU_P->CPG_Status = DONE;
             if ((PMU_P->CPG_Count==0) &&
                 (PMU_P->CPG_Ptr1 != NULL) &&
                 (PMU_P->CPG_Ptr1->Num_Prods != 0) &&
                 (PMU_P->CPG_Ptr2 != NULL) &&
                 (PMU_P->CPG_Ptr2->Num_Prods != 0)) {
                printf("\nEMPTY CLAUSE IN THE GPF:");
                printf(" PROCESS TERMINATED ABNORMALLY.");
                printf("\n       NO SOLUTION FOUND");
                printf("\nDevice#%2d:\nFirst Clause:",
                          Device_ID);
                Echo_a_Clause(PMU_P->CPG_Ptr1);
                printf("\nSecond Clause:");
                Echo_a_Clause(PMU_P->CPG_Ptr2);
                GPFS_Report();
                Normal_Termination = FALSE;
```

```
                HOST.STATUS = IDLE;
                HOST.Mail_Box = FINISH;
                } /* Empty clause: The GPF has no solution */
              } /* Complete Cartesian Generating process */
          }
      if ((GPFS[Device_ID].PMU->CPG_Status == DONE) ||
          (GPFS[Device_ID].PMU->CPG_Count >= SAPA_SIZ*2)){
         if (PMU_P->CPG_Ptr1 != NULL)
            Release_a_Clause (PMU_P->CPG_Ptr1);
         if (PMU_P->CPG_Ptr2 != NULL)
            Release_a_Clause (PMU_P->CPG_Ptr2);
         PMU_P->CPG_Ptr1 = PMU_P->CPG_Ptr2 = NULL;
         PMU_P->CPG_Ptr3 = PMU_P->CPG_Ptr4 = NULL;
         PMU_P->CPG_Count  = 0;
         PMU_P->CPG_Status = DONE;
         }
      return (Send_Prod);
        /* Returned to PMU to transfer to SPRU */
      } /* BUSY: Valid CPG clause pointers */
   else return (NIL_PROD);
} /* End CPG Operations */



Boolean Transfer_Clauses_to_CPG (Device_ID)
int Device_ID;
/*
   Hang a new pair of clauses to CPG pointers. Clauses
   transferred to the CPG are taken from the top of the list.
   While, the products generated from the CPG will be sent to
   the SPRU. The min-cost products returned from the SAPA
   (if not propagating to the next DFT level), will be linked
   back to the PMU linked list of clauses, at the end-of-list
   clause [Clau_Link, Prod_Link].  Initialize the Product
   pointers appropriately.  This function returns a SUCCESSFUL
   status if the transfer is completed. Otherwise, it returns
   FAILURE */
{
   Pmu_Ptr u;
   u = GPFS[Device_ID].PMU;
   if (u->CPG_Ptr1 != NULL)
      free( (Clau_Ptr) u->CPG_Ptr1);
   if (u->CPG_Ptr2 != NULL)
      free( (Clau_Ptr) u->CPG_Ptr2);

   /* If clauses available, feed the CPG with new pair
      from the BPP clause header. */
   if ((u->Clause_Header != NULL) &&
       (u->Num_Clau_Possess >= 2)) {
      u->CPG_Ptr1 = u->Clause_Header;
      u->CPG_Ptr2 = u->Clause_Header->Next;
      u->Clause_Header = u->Clause_Header->Next->Next;
```

```
      u->CPG_Ptr1->Next = u->CPG_Ptr2->Next = NULL;
      u->CPG_Ptr3 = u->CPG_Ptr1->Next_Prod;
      u->CPG_Ptr4 = u->CPG_Ptr2->Next_Prod;
      u->CPG_Count = 0;
      u->CPG_Status = BUSY;
      u->Num_Clau_Possess -= 2;
      return(SUCCESSFUL);
      }
   /* Otherwise, not enough clauses left for the CPG.
      Initialize all relevant pointers and return the
      FAILURE status. */
   else {
      u->CPG_Ptr1 = u->CPG_Ptr2 = NULL;
      u->CPG_Ptr3 = u->CPG_Ptr4 = NULL;
      u->CPG_Count = 0;
      u->CPG_Status = DONE;
      return(FAILURE);
      }
} /* End Transfer_Clauses_to_CPG */



Prod_Ptr PMU_Operations (Device_ID)
int Device_ID;
{
   Pmu_Ptr  PMU_P;
   Spru_Ptr SPRU_P;
   Prod_Ptr p = NIL_PROD;
   Prod_Ptr CPG_Operations();
   Boolean Transfer_Clauses_to_CPG();

   PMU_P  = GPFS[Device_ID].PMU;
   SPRU_P = GPFS[Device_ID].SPRU;
   GPFS[Device_ID].SEL_Sapa = FALSE;

   /* CPG ready for next pair of clauses */
   if ((PMU_P->CPG_Status == DONE) ||
       (PMU_P->CPG_Ptr1==NULL) || (PMU_P->CPG_Ptr2==NULL)) {
      if ((SPRU_P->SAPAL->EMPTY == TRUE) ||
          (SPRU_P->SAPAR->EMPTY == TRUE)) {
         GPFS[Device_ID].SEL_Sapa = TRUE;
         if (Transfer_Clauses_to_CPG(Device_ID) ==
             SUCCESSFUL)  p = CPG_Operations(Device_ID);

         /*  else: UnSUCCESSFUL in transfer clauses to the
             CPG: Out of data, request parents for more.*/
         } /* else: both SAPAs BUSY. Wait til one EMPTY */
      }   /* CPG: already setup. Proceed its operations */
   else p = CPG_Operations(Device_ID);
   return(p);
} /* End PMU Operations */
```

```
/*-------------------------------------------------*/
/* GENERALIZED PROPOSITIONAL FORMULA SOLVER - Part#4 */
/*-------------------------------------------------*/
/* LINK LIST VERSION.                              */
/* This program simulates the operation of the SAPA. */
/*-------------------------------------------------*/
/*     Filename:       SAPA.C                      */
/*     Created:        FRI  02 JUN 1989      20:00 */
/*     Last Edited:    FRI  13 OCT 1989      22:45 */
/*-------------------------------------------------*/



void Release_Product_Node (Node_Ptr)
Prod_Ptr Node_Ptr;
{
     free( (Prod_Ptr) Node_Ptr);
} /* End Release_Product_Node */



Prod_Ptr Create_Product_Node ()

/*  Allocate memory for a Product Node which contains
    a pointers pointing to next Product node, and an
    long word encoded product. */
{
     int i; /* Temporary variable */
     Prod_Ptr PRD_Ptr;

     if ((PRD_Ptr = (Prod_Ptr) malloc
                    (sizeof(Product_Node)))==NULL) {
        printf("\nErrors in creating product nodes.");
        exit(11);
        }
     /* Initialize the Product Node switch dashes
        with all bits set to 1
     */
     for (i=0; i<WORD_SIZ; i++)
        PRD_Ptr->Prod_Patt[i] = 0xFFFF;

     /* Initialize the Product Node pointers
     */
     PRD_Ptr->Next = NULL;
     return ((Prod_Ptr) PRD_Ptr);
} /* End Create_Product_Node */
```

```
Sapa_Ptr Create_Sapa()
{
   int i,j;
   Sapa_Ptr Sp;
   Prod_Ptr Pp;
   Prod_Ptr Create_Product_Node();

   if ((Sp = (Sapa_Ptr) malloc
             (sizeof(Sapa_Node))) == NULL) {
      printf("\nErrors in creating SAPA nodes.");
      exit(11);
      }
   /* First pair of P-Q nodes with Empty Product Pattern
   */
   Sp->Q_Reg = Create_Product_Node();
   Sp->P_Reg = Create_Product_Node();

   for (i=1; i<SAPA_SIZ; i++) {
      Pp = (Prod_Ptr) Create_Product_Node();
      Pp->Next  = Sp->Q_Reg;
      Sp->Q_Reg = Pp;
      Pp = (Prod_Ptr) Create_Product_Node();
      Pp->Next  = Sp->P_Reg;
      Sp->P_Reg = Pp;
      }
   Sp->In_Cnt = Sp->Out_Cnt = 0;
   Sp->EMPTY = TRUE;
   return((Sapa_Ptr) Sp);
} /* End Create_Sapa */



Spru_Ptr Create_SPRU()
{
   Spru_Ptr S;
   Sapa_Ptr Create_Sapa();

   if ((S = (Spru_Ptr) malloc
            (sizeof(Spru_Node))) == NULL) {
      printf("\nErrors in creating SPRU nodes.");
      exit(11);
      }
   S->SAPAL = (Sapa_Ptr) Create_Sapa();/* Left  SAPA */
   S->SAPAR = (Sapa_Ptr) Create_Sapa();/* Right SAPA */
   S->Switch = LEFT;    /* Selector: Left/Right SAPA */
   return( (Spru_Ptr) S);
} /* End Create SPRU */
```

```
void SAPA_Reset(Sapa_P)
Sapa_Ptr Sapa_P;
{
   int i, j;
   Sapa_Ptr S;
   Prod_Ptr P,Q;

   P = Sapa_P->P_Reg;
   Q = Sapa_P->Q_Reg;

   for (i=0; i<SAPA_SIZ; i++) {
   /* Initialize the Product Node switch dashes
      with all bits set to 1: Init Pattern.
      16 Elements, each of them is 16 bit word.
   */
      for (j=0; j<WORD_SIZ; j++)
         *(P->Prod_Patt+j) = *(Q->Prod_Patt+j) = 0xFFFF;
      P = P->Next;
      Q = Q->Next;
      }
   Sapa_P->In_Cnt  =  Sapa_P->Out_Cnt = 0;
   Sapa_P->EMPTY = TRUE;
   if (DEBUGS==TRUE) printf("\n\tSAPA:---Reset---");
} /* End SAPA: Reset */



Prod_Ptr SAPA_Indicate_Dominating_Product
                        (P_Ptr, Q_Ptr, P_Cost, Q_Cost)

Prod_Ptr P_Ptr, Q_Ptr; /* Pointers to 2 given products */
Element  P_Cost, Q_Cost;

/*  This function will check for product domination
    property and will return the Prod_Ptr pointing
    to the dominating product (to be deleted). If no
    dominating product is detected, the function will
    return a NULL pointer value. */
{
   int i;
   E_Ptr p, q;
   void Echo_a_Product();

   p = P_Ptr->Prod_Patt;
   q = Q_Ptr->Prod_Patt;

   if ((P_Cost >= 0x7F00) ||
       (P_Cost == 0x0000)) { /* Clear Tag Bit */
      *(P_Ptr->Prod_Patt+(WORD_SIZ-1)) &= 0x7FFF;
      return(P_Ptr);
      }
```

```
    else if ((Q_Cost >= 0x7F00) ||
         (Q_Cost == 0x0000)) { /* Clear Tag Bit */
       *(Q_Ptr->Prod_Patt+(WORD_SIZ-1)) &= 0x7FFF;
       return(Q_Ptr);
       }
    else {
       for (i=0; i<WORD_SIZ; i++)
         *(MAP_PROD->Prod_Patt+i) =
                    *(p+i) | *(q+i);/* Bitwise OR */
       if ((P_Cost <= Q_Cost) &&
           ((i=memcmp((char*) MAP_PROD,
                      (char*) p,
                     WORD_SIZ*2-1)) == 0)) {
           *(Q_Ptr->Prod_Patt+(WORD_SIZ-1)) &= 0x7FFF;
            return(Q_Ptr);
            /* Reset tag bits, check out the 1st product */
            }
       else if ((i=memcmp((char*) MAP_PROD,
                          (char*) q,
                         WORD_SIZ*2-1)) == 0) {
          *(P_Ptr->Prod_Patt+(WORD_SIZ-1)) &= 0x7FFF;
          return(P_Ptr);
          /* set tag bits to check out the 2nd product */
          }
       return(NULL);
              /* no domination, reserve both products */
       }
} /* End SAPA: Indicate Dominating Product */


void SAPA_Compare_Swap_Tag(Sapa_P)
Sapa_Ptr Sapa_P;
{
   int i;
   Element Q_cost, P_cost;
   Prod_Ptr Ph, Pt, Qh, Qt, tp;
   Prod_Ptr SAPA_Indicate_Dominating_Product();

   Ph = Sapa_P->P_Reg;
   Qh = Sapa_P->Q_Reg;

   for (i=0; i<SAPA_SIZ; i++) {
      /* Get the costs. Mask out the Tag and Product Fields
      */
      P_cost = *(Ph->Prod_Patt + (WORD_SIZ-1)) & 0x7F00;
      Q_cost = *(Qh->Prod_Patt + (WORD_SIZ-1)) & 0x7F00;

      /* Tag Dominating Product: Clear MSB to check out
      */
      tp = SAPA_Indicate_Dominating_Product
                                 (Ph, Qh, P_cost, Q_cost);
```

```
      if ((DEBUGS==TRUE) && (tp != NULL)) {
           printf("\n\tDominating Product Detected.");
           Echo_a_Product(Ph);
           Echo_a_Product(Qh);
           }
      if (Q_cost < P_cost) { /* swap pointers */
         if ((Ph->Next != NULL) &&
             (Qh->Next != NULL)) {  /* not EOL */
            tp = Ph->Next;
            Ph->Next = Qh->Next;
            Qh->Next = tp;
            }
         /* else: end of list node,
                  no need to reroute pointers
         */
         if (i==0) {
            /* header node, to be handled differently */
            Sapa_P->P_Reg = Qh;
            Sapa_P->Q_Reg = Ph;
            } /* first node */
         else {
            Pt->Next = Qh;
            Qt->Next = Ph;
            }
         /* exchange the pointers so that Qh and Qt
            are always associated with Q_Reg;
            and Ph and Pt with P_Reg.
         */
         tp = Qh;
         Qh = Ph;
         Ph = tp;
         } /* Swap P & Q */
      Pt = Ph;
      Qt = Qh;
      Ph = Ph->Next;
      Qh = Qh->Next;
      } /* Check all PE registers */
} /* SAPA: Compare_Swap_Tag */


void SAPA_Input_New_Item (Sapa_P, Prod_P)
Sapa_Ptr Sapa_P;
Prod_Ptr Prod_P;
/*
   Check if the SAPA In_Cnt does not exceed the SAPA SIZE.
   IF EXCEEDED, reset the Sapa. Assume the CPG has to check
   for contradictions, and to compute the product costs
   before shifting them to the SAPA. */
{
   int i;
   Prod_Ptr Qp, QQ;
```

```
   void Release_Product_Node();
   void SAPA_Compare_Swap_Tag();

   /* Check the given product. Reject the NULL, Dominating,
      or clause terminator.
   */
   if ((Prod_P != MAX_PROD) && (Prod_P != NIL_PROD) &&
       (Prod_P != NULL) && (Sapa_P->In_Cnt<=SAPA_SIZ*2)) {
      /* Remove the end node */
      Qp = Sapa_P->Q_Reg;
      for (i=1; i<SAPA_SIZ; i++) {
          QQ = Qp;
          Qp = Qp->Next;
          }
      Release_Product_Node (Qp);
      QQ->Next = NULL;

      /* Link the given node in */
      Prod_P->Next = Sapa_P->Q_Reg;
      Sapa_P->Q_Reg = Prod_P;
      Sapa_P->In_Cnt++;
      Sapa_P->EMPTY = FALSE;
      SAPA_Compare_Swap_Tag(Sapa_P);
      }
} /* SAPA: Input A New Item */



Prod_Ptr SAPA_Output_Sorted_Item (Sapa_P)
Sapa_Ptr Sapa_P;

/* Check if the specified SAPA is in output state
   (Empty = False), and the number of Out_Cnt does
   not exceed a pre-specified limit.  IF EXCEEDED,
   reset the SAPA. The dominating (tagged) products
   are rejected and are not counted as output items.
*/
{
   int i;
   Prod_Ptr Pp;
   Prod_Ptr Create_Product_Node();
   void Release_Product_Node();
   void SAPA_Reset();
   void SAPA_Compare_Swap_Tag();

   /* No product exists in the SAPA
   */
   if (Sapa_P->EMPTY == TRUE) return(NULL);
   else { /* Valid product returned */
      SAPA_Compare_Swap_Tag(Sapa_P);
```

```
        /* Input an INIT Product Node (MAX),
           attatched to EOL (end of linked list)
        */
        if ((Pp = Sapa_P->P_Reg) != NULL) {
           i = 0;
           while ((Pp->Next!=NULL) && (i++<SAPA_SIZ))
              Pp = Pp->Next;
           Pp->Next = Create_Product_Node();
           }
        else {
           printf("\nERROR: SAPA HEADER LOST !! ");
           exit(22);
           }
        /* Unlink the output node. Return its address
        */
        Pp = Sapa_P->P_Reg;
        Sapa_P->P_Reg = Sapa_P->P_Reg->Next;
        Pp->Next = NULL; /* Detach the returned node */

        /* Check output product Tag field:
           return NIL_PROD if tag bits are cleared.
           Otherwise, increment output counter and reset
           the system if exceed required number
           (Limit the number of products in the return
           clause to be SEL_PRODS).
        */
        if ((Sapa_P->Out_Cnt >= SEL_PRODS) ||
            (Sapa_P->In_Cnt-- <= 0)) {
           Release_Product_Node(Pp);
           Sapa_P->EMPTY = TRUE;
           SAPA_Reset(Sapa_P);
           return(MAX_PROD);
               /* Empty: terminate the returned clause */
           }
        else if ((*(Pp->Prod_Patt+(WORD_SIZ-1)) & 0x8000)
                  == 0x0000) { /* Tag bit is cleared */
           if (DEBUGS==TRUE)
              printf("\n\tDominating Product Returned.");
           Release_Product_Node(Pp);
           return(NIL_PROD); /* Dominating product */
           }
        else { /* Good product node returned */
           Sapa_P->Out_Cnt++;
           return (Pp);
           }
        } /* SAPA is not EMPTY */
} /* SAPA: Output A Sorted Item */
```

```
Prod_Ptr SPRU_Operations (Device_ID, New_Prod)
int Device_ID;
Prod_Ptr New_Prod;
{
   Spru_Ptr Sp;
   Prod_Ptr Ret_Prod;
   Sapa_Ptr Left_Sapa, Right_Sapa;
   void SAPA_Reset();
   void SAPA_Input_New_Item();
   Prod_Ptr SAPA_Output_Sorted_Item();

   Sp = GPFS[Device_ID].SPRU;
   Left_Sapa  = Sp->SAPAL;
   Right_Sapa = Sp->SAPAR;

   if (DEBUGS==TRUE) {
      printf("\n\tSPRU: Input = ");
      if (New_Prod==MAX_PROD) printf(" %3s","MAX");
      else if (New_Prod==NIL_PROD) printf(" %3s","NIL");
      else printf("%#X,", New_Prod);
      }
   /* This section simulates the multiplexer function of the
      SPRU to select the Left or Right Sapa as Input or Output
      device at a time. Selector line is Alternate_Sapa.
   */
   if (GPFS[Device_ID].SEL_Sapa == TRUE) {
      if ((Sp->Switch == LEFT) &&
          (Right_Sapa->EMPTY == TRUE)) {
         /* Left SAPA is the current INPUT unit: Make it an
            OUTPUT unit. Right SAPA is the present OUTPUT
            unit: Stop the output stream, reset the unit, and
            set its status to accept new products.
         */
         Sp->Switch = RIGHT;
         GPFS[Device_ID].SEL_Sapa = FALSE;
         }
      else if ((Sp->Switch == RIGHT) &&
               (Left_Sapa->EMPTY == TRUE)) {
         /* Right SAPA is the current INPUT unit: Make it an
            OUTPUT unit. Left SAPA is the present OUTPUT unit:
            Stop the output stream, reset the unit, and set
            its status to accept new products. Now new items
            will enter the Left SAPA.
         */
         Sp->Switch = LEFT;
         GPFS[Device_ID].SEL_Sapa = FALSE;
         }
      } /* Alternate SAPA */

   if (Sp->Switch == LEFT) { /* Left=IN : Right=OUT */
      SAPA_Input_New_Item (Left_Sapa, New_Prod);
      Ret_Prod = SAPA_Output_Sorted_Item (Right_Sapa);
```

```
      }
    else {
    /* Loading Right SAPA. Outputing from Left SAPA */
       SAPA_Input_New_Item (Right_Sapa, New_Prod);
       Ret_Prod = SAPA_Output_Sorted_Item (Left_Sapa);
       }
  if (DEBUGS==TRUE) {
     if (Ret_Prod==MAX_PROD)
        printf("  Output = %3s","MAX");
     else if (Ret_Prod==NIL_PROD)
        printf("  Output = %3s","NIL");
     else printf(" Output = %#X",Ret_Prod);
     }
  return(Ret_Prod);
        /* MAX_PROD, NIL_PROD, NULL, or Prod_Ptr */
} /* End SPRU Operations */
```

```
/*-------------------------------------------------*/
/*     GENERALIZED PROPOSITIONAL FORMULA GENERATOR  */
/*-------------------------------------------------*/
/* This program simulates the operation of the GPFS. */
/* Version 2.0: 256 bit word with [Tag,Cost,Product] */
/*-------------------------------------------------*/
/*     Filename:       GENGPF.C                    */
/*     Created:        TUE  15 NOV 1988      15:00 */
/*     Last Edited:    SAT  14 OCT 1989      12:00 */
/*-------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void srand(seed);
   unsigned seed;
int rand();
FILE  *fopen(), *fp, *fpg, *fpp;


main(a1,a2)
int   a1;
char *a2[];
{
   static unsigned seed_rand;
   static int max_prod_size,
              max_clau_size,
              num_vars,
              num_clauses,
              neg_percent,
              literal;

   int tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7,
       tmp8, tmp9;
   long int  tmpa, tmpb;
   int buf[500], fn[80]; /* Not more than 500 literals */

   printf("\n OPEN DATAFILE: %12s", a2[1]);
   if ((fpp = (FILE *) fopen(a2[1],"r"))==(FILE *) NULL)
      printf("\n Sorry, cannot open file '%12s'.",a2[1]);

   else while (fscanf(fpp, "%s", fn) != EOF) {
      if ((fpg = (FILE *) fopen(fn,"w"))==(FILE *) NULL){
         printf("\n Sorry, cannot open file '%12s'.",fn);
         exit(11);
         }
      else fp = (FILE *) fopen("log.bak","w");

      /* User specifications.
      */
      fscanf(fpp, "%d",   &seed_rand);
```

```
fscanf(fpp, "%d",   &num_vars);
fscanf(fpp, "%d",   &max_prod_size);
fscanf(fpp, "%d",   &max_clau_size);
fscanf(fpp, "%d",   &num_clauses);
fscanf(fpp, "%d\n", &neg_percent);

if (neg_percent==0) {
   fclose(fp);
   fp = fpg;
   } /* No negated literals: no temp file */
tmpa = tmpb = 0;

fprintf(fp,"\n");
srand(seed_rand);

/* Generating Generalized Propositional Formula
   of NUM_CLAUSES.
*/
for (tmp1=0; tmp1<num_clauses; tmp1++) {
   while ((tmp0=abs(rand()%max_clau_size))==0);

   /* Generating a clause of TMP0 products.
   */
   for (tmp2=0; tmp2<tmp0; tmp2++) {
       while
          ((tmp4=abs(rand()%max_prod_size))==0);

       /* Generating a product of TMP4 literals.
          Literal must be Pos+  and <= NUM_VARS.
       */
       tmp7 = 0;
       while (tmp7<tmp4) {
          while ((tmp6=abs(rand()%num_vars))==0);

          /* Rejecting duplicate literals in a
             product.
          */
          if (tmp7==0) buf[tmp7++] = tmp6;
          else { /* 2nd to tmp4 literals */
             for(tmp8=tmp9=0; tmp8<tmp7; tmp8++)
                 if (tmp6==buf[tmp8]) tmp9=1;
             if (tmp9==0) buf[tmp7++] = tmp6;
             }
          } /* while tmp7: literals */

       /* Echo a product of literals from BUF out
          to file
       */
       for (tmp5=0;  tmp5<tmp7; tmp5++) {
          /* 10 literals per line
          */
          if ((tmp5 % 10) == 9)
```

```
                fprintf (fp, "%5d\n", buf[tmp5]);
              else fprintf (fp, "%5d", buf[tmp5]);
              }
            tmpa += (long int) tmp7; /* total number of
                 literals in all products in the GPF */
            fprintf (fp, "  999\n");
            } /* for tmp2: products */
         fprintf (fp, " 9999\n");
         } /* for tmp1: clauses */
      printf("\nFile '%12s' generated.",fn);
      fclose(fp);


      /* Calculating & Generating complemented literals
         according to the user specifications.
      */
      if (neg_percent!=0) {
         fp = (FILE *) fopen("log.bak","r");

         /* freq of negated literals
         */
         tmpb = (neg_percent<=50) ? neg_percent
                              : (abs(100-neg_percent));
         tmpb = (tmpa/100) * tmpb;
         tmpb = ((tmpb!=0) && (tmpb<tmpa))
                               ? tmpa/tmpb : 1;
         tmp8 = tmp9 = 0;
         while (fscanf(fp, "%d", &literal) != EOF) {
            if ( (literal!=0)
               &&(literal!=999)
               &&(literal!=9999)) {
               if (neg_percent>50) literal= -literal;
               if ((neg_percent!=100) &&
                   ((tmp8++ % tmpb)==0))
                  literal = -literal;
               if ((tmp9++ % 10) == 9)
                  fprintf (fpg, "%5d\n", literal);
               else fprintf (fpg, "%5d", literal);
               }
            else {
               fprintf(fpg, "%5d\n", literal);
               tmp9 = 0;
               }
            }
         fclose(fpg);/* newly created GPF */
         fclose(fp); /* temporary file */
         } /* no negated literals */
      } /* file created successfully */
   fclose(fpp);

} /* End MAIN */
```

```
/*-------------------------------------------------*/
/* GENERALIZED PROPOSITIONAL FORMULA SOLVER.       */
/* Include File                                    */
/*                                                 */
/* This include file contains type definitions,    */
/* constant definitions, and external variable     */
/* declarations to be included into 4 modules      */
/* of the Generalized Propositional Formula        */
/* program.                                        */
/*-------------------------------------------------*/
/*  Filename:       GPF.H                          */
/*  Created:        FRI  02 JUN 1989       20:00   */
/*  Last Edited:    fri  13 OCT 1989       20:15   */
/*-------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <ctype.h>
#include <math.h>



/**** C O N S T A N T S    D E F I N I T I O N S ****/

#define PROD_DELIM  999
#define CLAU_DELIM 9999

#define WORD_SIZ 16
      /* Number of Elements (i.e., 16 bit quantity)
         in a bitmap representation of a product:
         16 by 16 = 256 bits ==> max of 124 literals */

#define ELT_LEN  8
      /* Number of Literals per Element (2 bits per
         literal. Eight literals per 16-bit Element) */

#define MAX_LITS 124
       /* Maximum number of Literals per Product */

#define MAX_CLAUS 4
      /* Number of clauses a BPPs can accommodate
         in its local memory. */

#define MAX_NUM_DFTS 7
      /* Maximum number of DFT levels in the GPFS */

#define MIN_SAPA_SIZ  16
      /* Minimum number of PEs in a SAPA */

#define MAX_NUM_BPPS 127
```

```
      /* Maximum number of BPPs in the GPFS */

#define MAX_NUM_LEAF 64
      /* Maximum number of LEAF BPPs in the GPFS */

#define IDLE     0
#define ACTIVE   1

#define OFF      0
#define ON       1

#define FALSE    0
#define TRUE     1

#define RIGHT    0
#define LEFT     1

#define DONE     0
#define BUSY     1

#define FAILURE     0
#define SUCCESSFUL 1

#define LCOMM    444
#define RCOMM    555
#define COMMUN   666
#define CLEAR    777
#define FINISH   888



/*****    T Y P E     D E F I N I T I O N S     ******/

typedef char Boolean;
typedef int Literal;
typedef unsigned Element; /* 16 bit: 0-65535 */
typedef Element *E_Ptr;
typedef Literal *L_Ptr;

typedef struct Product {
      Element Prod_Patt [WORD_SIZ];
             /* Bitmap product: 16*16 = 256 bits */
      struct Product *Next;
             /* Neighbor Product node */
      } Product_Node;

typedef Product_Node *Prod_Ptr;

typedef struct Clause {
      int Num_Prods;
      Prod_Ptr Next_Prod;
      struct Clause *Next;
```

```
      } Clause_Node;

typedef Clause_Node *Clau_Ptr;

typedef struct {
      Boolean EMPTY;  /* True/False */
      int In_Cnt;     /* Number of Input received   */
      int Out_Cnt;    /* Number of Output returned  */
      Prod_Ptr Q_Reg; /* Linked list of Q registers */
      Prod_Ptr P_Reg; /* Linked list of P registers */
      } Sapa_Node;    /* SAPA structure */
typedef Sapa_Node *Sapa_Ptr;

typedef struct {
      Boolean Switch; /* Selector: Left/Right SAPA  */
      Sapa_Ptr SAPAL; /* Left SAPA */
      Sapa_Ptr SAPAR; /* Rite SAPA */
      } Spru_Node;    /* SPRU structure */
typedef Spru_Node *Spru_Ptr;

typedef struct {
      int   Num_Clau_Possess;
      int   Num_Clau_Transfer;
      Clau_Ptr Clause_Header;
      Clau_Ptr Rcvr_Ptr1;
      Prod_Ptr Rcvr_Ptr2;
      Clau_Ptr Clau_Link;
      Prod_Ptr Prod_Link;
      Boolean  CPG_Status;
      int      CPG_Count;
      Clau_Ptr CPG_Ptr1;
      Clau_Ptr CPG_Ptr2;
      Prod_Ptr CPG_Ptr3;
      Prod_Ptr CPG_Ptr4;
      } Pmu_Node;
typedef Pmu_Node *Pmu_Ptr;

typedef struct {
      Boolean   Status; /* Active / Idle */
      Boolean   SEL_Sapa;
                /* Alternate Sapa I/O functions */
      int       Mail_Box;
         /* Device_ID:   Son request pending.
            COMMUN  666: Fullfill/Clear the request.
                         Transfering data.
                         [LCOMM 444, RCOMM 555]
            FINISH  888: No more data in upper level.
            CLEAR   777: Clear request. */
      int       Num_Idles;
      int       Num_Clauses_Received;
      int       Left_ID;
      int       Rite_ID;
```

```
      int       Succ_ID;
      int       MAX_CLAU;
      Clau_Ptr  Xmit_Ptr;
         /* Clause to be sent to the next level */
      Pmu_Ptr   PMU;
         /* Product Management Unit Structure */
      Spru_Ptr  SPRU;
         /* Sum of Product Reduction Unit */
      } BPP_Node;
typedef BPP_Node *BPP_Ptr;

typedef struct {
   Boolean STATUS;   /* ACTIVE/IDLE */
   int   Mail_Box;
   int Total_Exec_Cycles; /* Total number of
                          execution steps required */
   int Num_Feedback_Clauses;
       /* Number of clauses received from the root */
   int Num_Clau_Possess;
           /* Number of clauses available for BPPs */
   int Num_Final_Prods;
               /* Num of products in the final SOP */
   Clause_Node *Leaf_Xfer [MAX_NUM_LEAF];
                     /* Clauses sent to the Leaf-node */
   Clau_Ptr Host_Ptr; /* Clauses in the original GPF */
   Clau_Ptr Rcvr_Ptr; /* Clauses from the root node BPP */
   Prod_Ptr SOP_Ptr;  /* Pointer to the final result  */
   } Big_Computer;


/* Bitmap MASKs for encoding the non-negated and
   negated literals
*/
Element POS_LITS[ELT_LEN] =
        {0xFFFE,0xFFFB,0xFFEF,0xFFBF,
         0xFEFF,0xFBFF,0xEFFF,0xBFFF};/* Lsb .. Msb */

Element NEG_LITS[ELT_LEN] =
        {0xFFFD,0xFFF7,0xFFDF,0xFF7F,
         0xFDFF,0xF7FF,0xDFFF,0x7FFF};

/* Bitmap patterns (MASKs) for checking contradiction
   products, and for calculating the product costs.
*/
Element CHK_MASK[ELT_LEN] =
        {0x0003,0x000C,0x0030,0x00C0,
         0x0300,0x0C00,0x3000,0xC000};


/********     G L O B A L     V A R I A B L E S   ********/

Prod_Ptr MAP_PROD;
```

```
    /* Product node used as temporary storage for
       checking product domination relation b/w two
       given products nodes. SAPA operations. */

Prod_Ptr MAX_PROD;
    /* Product node used as indicator for clause
       separator during data transfer between PMU/
       SAPA and BPP/BPP in the system. */

Prod_Ptr NIL_PROD;
    /* Product node used as Contradict/Dominating
       products which will not be included in the
       partial/final SOP expressions */

int SAPA_SIZ;
    /* Number of PEs in a SAPA. A SAPA of this size
       can sort a sequence of 2 * SAPA_SIZ products
    */
int SEL_PRODS;
    /* Number of products selected from a sorted
       clause to be transferred to the BPP in the
       next DFT level. This number is set to the
       square root of 2 * SAPA_SIZ
    */
int Host_ID;  /* Host index = Total BPPs */
int NUM_DFTS; /* Number of levels in the DFT */
int NUM_BPPS; /* Number of BPPs in the GPFS */
int NUM_LEAF; /* Number of Leaf-BPPs */
int LEAF_INX; /* First Leaf-node index */

int NUM_PORT; /* Number of Leaf-BPPs the Host can
                 serve  at every execution cycle */
int LAST_LEAF;/* Index of the last BPP in the leaf DFT
                 level being served by the Host */

BPP_Node GPFS[MAX_NUM_BPPS];
    /* The Generalized Propositional Formula Solver */
    /* E.g., Root node [0] connected to [1] and [2] */
    /* Node [1] is connected to nodes [3] and [4]   */
    /* Node [2] is connected to nodes [5] and [6]   */

Big_Computer HOST;
```

```
/*-------------------------------------------------*/
/* Program TIMER                                   */
/* Include File.                                   */
/* Measure program execution time, using the MS-DOS  */
/* system real-time clock.                         */
/*-------------------------------------------------*/
/*  Filename:       TIMER.H                        */
/*  Created:        FRI  02 JUN 1989       20:00   */
/*  Last Edited:    FRI  16 JUN 1989       23:45   */
/*-------------------------------------------------*/

#include <dos.h>
union REGS inreg, outreg;

double start_time, finish_time;
double get_time()
{
   unsigned int hr, min, sec, hun;
   double timing;

   inreg.h.ah = 0x2c;
   intdos (&inreg, &outreg);

   hr  = outreg.h.ch;
   min = outreg.h.cl;
   sec = outreg.h.dh;
   hun = outreg.h.dl;

   /* elapsed time in seconds.
   */
   timing = ((double) hr)  * 3600.0 +
            ((double) min) * 60.0 +
            ((double) sec) +
            ((double) hun) / 100.0;
   return (timing);
}

double how_long (start, finish)
double start,
       finish;
{
   return ((start > finish) ? 0.0
                            : (finish-start));
}
```

APPENDIX C

THE GPFS SIMULATION - TEST CASES AND RESULTS

```
##########################################################
          F I N A L    S O P    S O L U T I O N
##########################################################


   5   6   8  11  15  17 -18  21  22  23  24  25
  26 -30  31  33  35  37  38  41  43  44  51
  52  54  57  58  59  61  62  63  66  67  68
  70 -72  73  74  76  77  78  79  82  84  88
  89  90  91  92 -94  95  96 100 103 104 105
 106 107 109 110 112 113 114 115 118
[Product_Cost=  65]
   5   6   8   9  11  15  17 -18  21  22  23  24
  25  26  31  33  35  37  38  41  43  44  51
  52  54  57  58  59  61  62  63  66  67  68
  70 -72  73  74  76  77  78  79  82  84  88
  89  90  91  92 -94  95  96 100 103 104 105
 106 107 109 110 112 113 114 115 118
[Product_Cost=  65]
   2   5   6   8  11  15  17 -18  21  22  23  24
  25  26 -30  31  33  35  37  38  41  43  44
  51  52  54  57  58  61  62  63  66  67  68
  70 -72  73  74  76  77  78  79  82  84  88
  89  90  91  92 -94  95  96 100 103 104 105
 106 107 109 110 112 113 114 115 118
[Product_Cost=  65]
   2   5   6   8   9  11  15  17 -18  21  22  23
  24  25  26  31  33  35  37  38  41  43  44
  51  52  54  57  58  61  62  63  66  67  68
  70 -72  73  74  76  77  78  79  82  84  88
  89  90  91  92 -94  95  96 100 103 104 105
 106 107 109 110 112 113 114 115 118
[Product_Cost=  65]
   2   5   6   8  11  15  17 -18  20  21  22  23
  24  25  26 -30  31  33  35  37  38  41  43
  44  51  52  54  57  58  61  62  63  66  67
  68  70  73  74  76  77  78  79  82  84  85
  88  89  90  91  92  93 -94  95  96 100 103
 104 105 106 107 108 109 110 112 113 114 118
[Product_Cost=  67]
   2   5   6   8  11  15  17 -18  20  21  22  23
  24  25  26 -30  31 -32  33  35  37  38  41
  43  44  51  52  54  57  58  61  62  63  66
  67  68  70  73  74  76  77  78  79  82  84
  85  89  90  91  92  93 -94  95  96 100 103
 104 105 106 107 108 109 110 112 113 114 118
```

```
      [Product_Cost=  67]
         2    5    6    8   11   15   17 -18   20   21   22   23
        24   25   26 -30   31   33   35   37   38   41   43
        44   51   52   54   57   58   60   61   62   63   66
        67   68   70   73   74   76   77   78   79   82   84
        85   89   90   91   92   93 -94   95   96  100  103
       104  105  106  107  108  109  110  112  113  114  118
      [Product_Cost=  67]
         2    5    6    8   11   15   17 -18   20   21   22   23
        24   25   26 -30   31 -32   33   35   37   38   41
        43   44   48   51   52   54   57   58   59   61   62
        63   66   67   68   70   73   74   76   77   78   79
        82   84   85   89   90   91   92   93 -94   95   96
       100  103  105  106  107  108  109  110  112  113  114
       118
      [Product_Cost=  68]
         2    5    6    8   11   15   17 -18   20   21   22   23
        24   25   26 -30   31   33   35   37   38   41   43
        44   51   52   54   57   58   59   60   61   62   63
        66   67   68   70   73   74   76   77   78   79   82
        84   85   89   90   91   92   93 -94   95   96  100
       103  104  105  106  107  108  109  110  112  113  114
       118
      [Product_Cost=  68]
         2    5    6    8   11   15   17 -18   20   21   22   23
        24   25   26 -30   31   33   35   37   38   41   43
        44   51   52   54   57   58   59   61   62   63   66
        67   68   70   73   74   76   77   78   79   82   84
        85   88   89   90   91   92   93 -94   95   96  100
       103  104  105  106  107  108  109  110  112  113  114
       118
      [Product_Cost=  68]
[Num_Products = 10]
```

```
##########################################################
            FINAL REPORT ON BPPS PERFORMANCES
##########################################################

      DFT  = 2
      PORT = 2
      SAPA = 100

BPP# 2:
      Number of IDLE cycles =  1149
      Number of Clauses Possess =    0
      Number of Clauses Tranfer =   37
      Number of Clauses Received =   67
BPP# 1:
      Number of IDLE cycles =  1100
      Number of Clauses Possess =    0
      Number of Clauses Tranfer =   64
      Number of Clauses Received =   90
BPP# 0:
      Number of IDLE cycles =    55
      Number of Clauses Possess =    1
      Number of Clauses Tranfer =   79
      Number of Clauses Received =  122


##########################################################
          SUMMARY OF THE GPFS EXECUTION & RESULTS
##########################################################

DFT  IDLE  Receive  Transfer IDLE/ACTIVE
 0     55      122       79    2.70
 1   1124       78       50   55.13

HOST: Clauses returned from the DFT:  79
Optimal Product: Cost =   65
Total GPFS Execution Cycles:  2039
Elapsed Time = 90.130000 seconds.
```

```
##########################################################
                    D A T A F I L E
##########################################################

  -94  100   89  999
 9999
   78   82  104   65    1   27  999
   11  999
   62   27   36  999
   84   62   33   52   22   21  116  999
 9999
   47    6   11   18   69  112  107  999
  115   54   23  999
    2   93   33  999
   31   53   28  107  -44  999
  117   19   83   21   49   58  999
   35  110   42   48  999
 9999
   64   88   86   45   50    9  999
  110    6  999
 9999
   68  999
   23   84   34   36   80  999
   56   11   28   24   39   26  999
   17   58   38  999
    9  101  999
 9999
   79  -18   24  999
 9999
   26  999
   66  999
 9999
   44  999
   69   17   93   57   72  106  999
   41   36  999
   47   15   94  999
   52  110   70   21    4   46   27  999
 9999
   57   97   47   33   63   65   29  999
   38  999
  108   62  106   26   70  999
    8  -60   71   42   95  999
  119   24  999
   68  999
  115   81    2  999
 9999
  116   14  100  999
   21  108   39    4  999
```

```
  54  999
  79   18   98   60   68  999
 107    8   13   48   43   47   21  999
  17   13   34   29   56   95  999
9999
  79  999
-118   63  104   88  999
  89  109   62   75  999
9999
  63   43   27   34   23  999
  18   60  999
  38   41   29   49  999
  72  102   58   52   38  999
  70   17   38  999
9999
  88   76  111    2  114   32   15  999
  82   86   35 -113  109  102  999
  96   41   78    2  999
  61   59   77  999
  52   29   35   32  999
   3  109  101   48   81   69  999
 103   82   25   62   85   88  106  999
9999
 112   92   89  114   41  999
9999
  88  999
  65  115   74  109  999
 -32  999
  60  999
  33   59   63   79   76    7  999
9999
  37   66  109   73   35  999
9999
  25    8   82   95   54  114  999
9999
  31   26   23  113  999
 105   13   72  999
 113  999
   1  118    6   22  999
  64  999
9999
  49    1   15   50   13  -26  999
  59   12   39   93  999
 110   65  999
 106  999
9999
  62    8  999
9999
  65   78   71  999
  25   77  999
9999
  24   54  999
```

```
  12  119  999
 110  999
  97   78   17   13  110   83   28  999
 102  999
  67    2   19   47  105  999
9999
 103   92  101   21  999
 -72  105  113  999
  71   92   22   62   86  999
 107   80   93   14  999
  10  115  113   31   80   56  999
  85   70   20  999
  55   24   59   85   56   94  999
9999
  91   26  118   33   44   58  999
 112   40   87  100   -6   10   77  999
9999
  96  107   43  999
  84   69   82   46   45    7  999
9999
 105   67  999
9999
  80   62   82    3  116   17   61  999
   5   64  100  999
  76  110  999
  91   51  107   33   40   30  999
  68   43   20   91  999
9999
  60  -73  118   58   85  999
  77   18   79   23   40   37   44  999
  68  113    7    1   50  999
 108   33   24   85  999
  31   24  115   76  999
 103   85   36   56   66  117  999
9999
  57  111  109   81   56  999
  97   32 -117   96   65   41  999
   9  109    5   79   52   96   95  999
  42  104   94   54   52  999
  89  999
  83   87   71  115  999
9999
  98   63   62  999
  31   61  999
  99   65  101  110   66   14   85  999
9999
 118   79  999
  73   14  -77   36   96  999
   2  999
  41   60   13   81  999
  15   62   11   37   24   38   72  999
  96   73   80  999
```

```
  35  999
9999
  57   52  999
  89   71   88    5   10   57   69  999
9999
  33  999
  51  100  999
  98   95  119   32  999
  64   61  -63  109   15  999
 117   29   36    1  999
 107   21   89  999
9999
  81   77   55   43   53  112  999
 112  999
  30   28   67    4  115   66   18  999
  47   87  999
  23  101  103   66  999
9999
  91    9  999
 102   59   33   31  999
 112  -30  999
9999
  99   63   22   98   87   24   88  999
 104  113   35  999
  32  999
  78   89  999
 119   65   89  104  105   60  999
9999
 118   30    1    3  999
 112   32    7   10  999
  21   63   23  999
  80   89   95   45   28   63  -27  999
  37   89   63    9    1   28   62  999
9999
  78   74    8  106   90  999
   3   70   60   28   67  999
  35   66   65   78   13   69   64  999
   3  119   97   46  999
9999
  86   91   25   75   41  109  999
  84   62  -35  999
  62  999
9999
 112   26   86   13  999
   4  105  106   52   55  999
   5   76   96  101   94  999
  96   52  999
9999
  35   15  999
 110   81  105  999
 117  106   10  999
  22   34  999
```

```
  33   56   12   88   62  999
  54  999
9999
  23   96  999
  84  -77   13  109  999
  40   80   78   58  999
  35   41  999
  76    3   29   22   81   49  108  999
  93   74  999
   2   83  111   49  999
9999
  48   59   91  999
 105   61  104  999
 113   98   69    2   58  999
  50   21  114   28   -1  999
  13   75   69   97   58   60   51  999
  15   75   90   39   92  999
9999
   5   70  999
  54  100   55   31  999
9999
 119   70   98  999
  68  999
  12  999
   9   73   48  101  999
  43    3   28  104   58   31  999
9999
 111   53   67 -113   15    8  999
  76  999
  90   78   69   13    6  999
   9   27   88   39  999
   5   66   96   76    6  999
9999
  81   44   20   46   21   22   64  999
  15   22  113   84   61  999
9999
 103   51  999
9999
  78   82  107   67  999
  52  999
-114   49    4   36   50   53  999
 112   94   25   68  999
  32   91  114  999
  38    2   64   35   92   69   39  999
9999
  78   66   19   98   46   76   22  999
  38  999
  66   32   77   85   87   58  999
  87   18  999
  12  -80   78   39  112   48  999
  60  115  999
9999
```

APPENDIX D

THE CPS ALGORITHM - PROGRAM LISTING

```
/*-------------------------------------------------------*/
/*  Program COVERING PROBLEM SOLVER (CPS)      Part#1.*/
/*  LINK LIST VERSION.                                   */
/*-------------------------------------------------------*/
/*      Created:         MON  26 SEP 1988        17:00 */
/*      Last Edited:     THU  22 JUN 1989        18:50 */
/*-------------------------------------------------------*/

#include "TABCOV.H"
#include "TIMER.H"


/******    E X T E R N A L   F U N C T I O N S   *******/

extern int Enter_Rows(Implicant Symbol, int SW_Inx);
extern void Enter_Column(int COL_Inx, int SW_Inx);
extern int Table_Reduction();
extern void Release_All_Literal_Nodes();
extern void Reconstruct_Implicant_Nodes();
extern A_Ptr Create_SOL_Finder();
extern int SOL_Generator();


/*********    G L O B A L       V A R I A B L E S   *******/

Header CPS; /* Reference Node:
               contains head pointers & table counters */
int NF; /* Number of Literals */
int NL; /* Number of Implicants */
int LC; /* Length of Column strip [in byte: 16 bits]   */
int LR; /* Length of Row    strip [in byte: 16 bits]   */

int MAX_COST_ACCEPTABLE;
int NUM_SOLS_WANTED;
int Execution_Option;
int Output_Option; /* Flag controls results displayed */

Element Maski[ELT_LEN]   = {0x0001,0x0002,0x0004,0x0008,
                            0x0010,0x0020,0x0040,0x0080,
                            0x0100,0x0200,0x0400,0x0800,
                            0x1000,0x2000,0x4000,0x8000};
                            /* Lsb .. Msb */
Element Masko[ELT_LEN+1] = {0x0000,
                            0x0001,0x0003,0x0007,0x000F,
                            0x001F,0x003F,0x007F,0x00FF,
                            0x01FF,0x03FF,0x07FF,0x0FFF,
                            0x1FFF,0x3FFF,0x7FFF,0xFFFF};
                            /* Lsb .. Msb */
Implicant In_Buf[MAX_SIZ];
FILE *fp;     /* pointer to input data file. */
```

```
/****************    P R O C E D U R E S    ****************/


void Print_Line(chr1, chr2, Len)
/*
   This function prints a line of length 'Len', consists
   of the given character 'chr2'. The function also allow
   blank lines to be inserted, if desired, by sending a
   newline character for the 1st argument 'chr1'.
   The line length is adjusted accordingly.
*/
int  chr1, chr2, Len;
{
    int i, j;
    for (i=0; i<3; i++) putchar(chr1);
    j = (chr1==chr2)  ? Len-4  : Len;
    for (i=0; i<j; i++) putchar(chr2);
    putchar(chr1);
}



int New_Implicant(In_Dat)
/*
   Function returns insertion index if the newly acquired
   Implicant is not currently existed in the input buffer,
   In_Buf, where all recorded Implicants are organized in
   ascending order.  NIL returned, otherwise.
   The function is invoked by Exam_Input_Data().
 */
Implicant In_Dat;
{
   int i;
   for (i=0;i<NL;i++) {
      if (In_Dat==In_Buf[i])  return(NIL);
      else if (In_Dat<In_Buf[i]) return(i);
      }
}/* End New_Implicant */



void Insert_New_Implicant(Index,In_Dat)
/*
    This function inserts the given input item into the
    In_Buf array at location specified by the passing Index.
 */
int Index;
Implicant In_Dat;
{
   int i;
```

```
   for (i=NL;i>Index;i--) {
      In_Buf[i] = In_Buf[i-1]; /* Shift right 1 place */
      }
   In_Buf[Index] = In_Dat;

}/* End Insert_New_Implicant */




void Exam_Input_Data()
/*
   Examine given data in the specified text file. The
   Literals are separated by NIL constant.  Each Implicant
   will be represented as an integer, separated by blank(s).
 */
{
   char  fn[80]; /* Filename, less than 80 characters. */
   FILE *fopen();/* Library function */
   static Implicant In_Dat;
   int i;

   NF = 0; /* Initialize number_of_Literal  counter */
   NL = 0; /* Initialize number_of_Implicant counter */
   for (i=0;i<MAX_SIZ;i++)
      In_Buf[i] =  NIL; /* Init Input Buffer */

   printf("\n\t\tEnter Name of Data File: ");
   scanf("\n%s",fn);  /* Get  the filename */
   printf("%s\n",fn); /* Echo the filename */

   if ((fp = fopen(fn,"r")) == (FILE *) NULL) {
      printf("\n SORRY!...'%12s' NOT EXIST!\n",fn);
      exit(88);
      }
   else { /* Data file exists
             read the input data items */
      while (fscanf(fp,"%d",&In_Dat) != EOF) {
         if (In_Dat==NIL) NF++;
         else
         if ( (i = New_Implicant(In_Dat)) != NIL ) {
            NL++;
            Insert_New_Implicant(i,In_Dat);
            }
         }/* while not eof */

      LC = NL/ELT_LEN + (NL%ELT_LEN!=0);
      LR = NF/ELT_LEN + (NF%ELT_LEN!=0);

      if (Output_Option>=ON) {
         Print_Line('\n','#',79);
```

```
            printf("\nLIST OF IMPLICANTS:\n\t");
            for (i=0;i<NL;i++) {
                printf("%6d",In_Buf[i]);
                if (i%8==7) printf("\n\t");
                }
            printf
            ("\n\n\n\tNumber of Implicants = %5d",NL);
            printf
            ("\n\tNumber of Literals       = %5d",NF);
            printf("\n\n\tSize of Switch_Dash");
            printf(" [in 16-bit-word unit]");
            printf("\n\t\tImplicant Switch_Dash");
            printf(": %3d words.",LR);
            printf("\n\t\tLiteral   Switch_Dash");
            printf(": %3d words.",LC);
            }
        } /* Datafile exists */
    rewind(fp);
} /* End Exam_Input_Data */



void Get_Input_Data()
/*
    Get the input data from the given file and enter them
    into the table, at the appropriate location in each
    Row and Column. The dimension of this table is created
    by dynamically allocate memory according to the nature
    of the input entries extracted from the datafile by
    procedure  Exam_Input_Data.
 */
{
    int i,j,jj,k;
    static Implicant In_Dat;
    int Enter_Rows(Implicant Symbol,int SW_Inx);
    void Enter_Column(int COL_Inx,int SW_Inx);

    if (fp==NULL) printf("\n\t\tLOST DATAFILE !!!!!");
    else {
        i = 0; /* count of Implicants in the current row */
        j = jj = 0;
              /* indices to the current column (Literal) */
        if (Output_Option>=ON) {
            Print_Line('\n','#',79);
            printf("\nECHO DATAFILE:");
            printf("\n\tLiteral#%3d:",j++);
            }
        while ((fscanf(fp,"%d",&In_Dat) != EOF) &&
               (i++<MAX_SIZ)) {
            if (In_Dat==NIL) { /* end of current Literal */
                i = 0;
```

```
            if ((j<NF) &&
                (Output_Option>=ON))
               printf("\n\tLiteral#%3d:", j++);
            jj++;
            }
         else { /* enter Implicants into the table */
            if ((k = Enter_Rows(In_Dat,jj)) != NIL)
               Enter_Column(jj,k);
            else printf
                 ("\Error: enter data into table row ");
            if (Output_Option>=ON)
               printf ("%5d",In_Dat);
            }
         }
      } /* Valid file pointer */
      fclose(fp);
} /* End Get_Input_Data */



void Create_Essential_Literal_Node(Symbol)
/*
    Allocate memory for a node which contains the given
    Implicant and a pointer to the next possible node. These
    Implicants must appears in the resultant products.
 */
Implicant Symbol;
{
     int i; /* Temporary variable */
     S_Ptr Sav_Ptr, New_Ptr;

     /* Create & Init the Essential Implicant Node */
     New_Ptr = (S_Ptr) malloc (sizeof(Essential_Literal));
     New_Ptr->Next = NULL;
     New_Ptr->IMP = Symbol;

     if (CPS.ESS_Cnt==0)
        CPS.ESS_Ptr = New_Ptr; /* First node */
     else { /* Consecutive nodes */
        Sav_Ptr = CPS.ESS_Ptr;
        for (i=1; i<CPS.ESS_Cnt; i++)
           Sav_Ptr = Sav_Ptr->Next;
        Sav_Ptr->Next = New_Ptr;
        }
     CPS.ESS_Cnt++;

} /* End Create_Essential_Literal_Node */
```

```
F_Ptr Create_Table_Column()
/*
    Allocate contiguous memory for a Literal strip, which
    contains two pointers; one points to the next Literal
    strip node;  the other points to an array of bytes
    which conforms a bit array of the length of number of
    available Implicants. The starting address of the
    newly allocated block of memory will be returned to
    the calling routine.
 */
{
     int i; /* Temporary variable */
     F_Ptr Literal_Ptr;

     Literal_Ptr = (F_Ptr) malloc
                   (sizeof(Literal_Strip));
     Literal_Ptr->SW_Dash =
                   (E_Ptr) calloc(LC,sizeof(Element));

     /* Initialize the Literal strip */
     for (i=0;i<LC;i++)
        *((Literal_Ptr->SW_Dash)+i) = 0x0000;
     Literal_Ptr->Next = NULL;
     Literal_Ptr->Cnt = 0;

     if (Literal_Ptr==NULL) exit(77);
     return ((F_Ptr) Literal_Ptr);
} /* End Create_Table_Column */




L_Ptr Create_Table_Row()
/*
    Allocate contiguous memory for a Implicant strip,
    which contains an integer value represents for a
    Implicant of the given function and two pointers;
    one points to the next Implicant strip node; the
    other points to an array of bytes which conforms
    a bit array of the length of number of available
    Product of Sum Literals. The starting address of
    the newly allocated block of memory will be passed
    back to the calling routine.  */
{
     int i;
     L_Ptr Implicant_Ptr;

     Implicant_Ptr = (L_Ptr) malloc
                     (sizeof(Implicant_Strip));
     Implicant_Ptr->SW_Dash = (E_Ptr) calloc
                              (LR,sizeof(Element));
```

```
    /* Initialize the newly defined Implicant node */
    for (i=0;i<LR;i++)
       *((Implicant_Ptr->SW_Dash)+i) = 0x0000;
    Implicant_Ptr->Next = NULL;
    Implicant_Ptr->IMP  = NIL;

    if (Implicant_Ptr==NULL) exit(88);
    return ((L_Ptr) Implicant_Ptr);
} /* End Create_Table_Row */


void Create_CPS()
/*
    Allocate contiguous memory for the array of Literal
    strips and Row strips which together define a state
    table that corresponding to the given POS function
    specified in the input datafile. The Columns of the
    table contains the existence of the Implicants in
    each Literal, while each Row represent the state of
    an Implicant accross all available Literal.  */
{
     int i;
     F_Ptr p,pp;
     L_Ptr q,qq;

     /* Init CPS counters */
     CPS.COL_Cnt = NF;
     CPS.ROW_Cnt = NL;

     /* Create first Literal & Implicant Nodes */
     CPS.COL_Ptr = Create_Table_Column();
     CPS.ROW_Ptr = Create_Table_Row();
     p = pp = CPS.COL_Ptr;
     q = qq = CPS.ROW_Ptr;
     if ((pp==NULL) || (qq==NULL)) exit(44);

     /* Create successor nodes */
     for (i=1; i<NF; i++) {
         p = Create_Table_Column();
         pp->Next = p;
         pp = p;
         if (pp==NULL) exit(66);
         }
     for (i=1; i<NL; i++) {
         q = Create_Table_Row();
         qq->Next = q;
         qq = q;
         if (qq==NULL) exit(55);
         }
} /* End Create_CPS */
```

```
void Init_CPS()
{
     int i;
     F_Ptr p;
     L_Ptr q;

     CPS.PMP_Ptr = NULL;
     CPS.PMP_Cnt = 0;

     /* Initialize Save Pointer & Counter */
     CPS.ESS_Ptr = NULL;
     CPS.ESS_Cnt = 0;

     CPS.SOL_Ptr = NULL;

     /* Initialize the Literal node counters */
     i = 0;
     p = (F_Ptr) CPS.COL_Ptr;
     while ((i<NF) && (p!=NULL)) {
        p->Cnt = 0;
        p->FAC = i++;
        p = p->Next;
        }
      /* Initialize the Implicants in table ROWs */
     i = 0;
     q = (L_Ptr) CPS.ROW_Ptr;
     while ((i<NL) && (q!=NULL)) {
        q->IMP = In_Buf[i++];
        q = q->Next;
        }

}/* End Init_CPS */


void Architecture_Report()
/*
    This function prints out all pointer values and
    the contents of the allocated memory for the
    Implicant and Literal nodes. */
{
     int i,j;
     F_Ptr p;
     L_Ptr q;

     printf("\n\n\n\nARCHITECTURE REPORT:");
     printf("\n\tHeader Node: Column Cnt = %d\tRow Cnt = %d",
        CPS.COL_Cnt, CPS.ROW_Cnt);
     printf("\n\t              Col_Ptr=%4.4X",CPS.COL_Ptr);
     printf("\n\t              Row_Ptr=%4.4X",CPS.ROW_Ptr);
```

```
      j = 0;
      p = (F_Ptr) CPS.COL_Ptr;
      printf("\n\tLiteral Nodes:");
      while (p!=NULL) {
         printf("\n\t  Node#%3d  CNT=%3d   Addr=%8.8X  SW: ",

                j++,p->Cnt,p);
         for (i=0;i<LC;i++)
            printf("%4.4X  ",(Element) *(p->SW_Dash+i));
         p = p->Next;
         }
      j = 0;
      q = CPS.ROW_Ptr;
      printf("\n\n\tImplicant Nodes:");

      while (q!=NULL) {
         printf("\n\t  Node#%3d  IMP#%3d  Addr=%8.8X  SW: ",
                j++, q->IMP, q);
         for (i=0;i<LR;i++)
            printf("%4.4X  ",(Element) *(q->SW_Dash+i));
         q = q->Next;
         }
}/* End Architecture_Report */


void Switch_Box_Report(sema)
/*
    This function prints out the switching table.
*/
int sema;
{
      int i,j,k;
      F_Ptr p;
      L_Ptr q;

      if ((CPS.ROW_Cnt>0) && (CPS.COL_Cnt>0)) {

         Print_Line('\n','-',79);
         printf("                              ");
         printf("S W I T C H I N G    T A B L E\n");

         /*** PRINT Literal NODES ___ TABLE COLUMNS. ***/
         if ((sema==1) || (sema>=3)) {
            printf("                              ");
            printf("  IMPLICANTS versus LITERALS\n");
            Print_Line('-','-',79);
            p = (F_Ptr) CPS.COL_Ptr;
            while (p!=NULL) {
               j = 0;
               printf("\nLiteral #%4d:    ",p->FAC);
               for (i=0;i<LC;i++) {
```

```
            if ((ELT_LEN*i)<CPS.ROW_Cnt) {
               for (k=0;k<ELT_LEN;k++) {
                  if (++j>CPS.ROW_Cnt)
                     goto end1; /* Emergency exit */
                  if ((*(p->SW_Dash+i) &
                     Maski[k])!=0) printf("1");
                  else printf("0");
                  if (k==7) putchar(' ');
                  } /* for k */
               putchar(' ');
               } /* if ELT_LEN*i */
            }/* for i */
         end1:
         p = p->Next;
         } /* while */
      /* Print the Implicants */
       i = 0;
       q = CPS.ROW_Ptr;
       putchar('\n');
       Print_Line('.','.',79);
       printf("\nImplicants:");
       while (q!=NULL) {
          printf("%4d",q->IMP);
          q = q->Next;
          i++;
          if ((i>0) && ((i%8)==0))
             printf("\n         ");
          }
       } /* if sema */

   /*** PRINT IMPLICANT NODES ___ TABLE ROWS. ***/
   if ((sema==2) || (sema>=3)) {

      printf("                              ");
      printf("Implicants versus Literals\n");
      Print_Line('-','-',79);
      q = CPS.ROW_Ptr;
      while (q!=NULL) {
         j = 0;
         printf("\nImplicant#%4d:    ",q->IMP);
         for (i=0;i<LR;i++) {
            if ((ELT_LEN*i)<CPS.COL_Cnt) {
               for (k=0;k<ELT_LEN;k++) {
                  if (++j>CPS.COL_Cnt)
                     goto end2; /* Emergency exit */
                  if ((*(q->SW_Dash+i)&Maski[k])!=0)
                      printf("1");
                  else printf("0");
                  if (k==7) putchar(' ');
                  } /* for k */
               putchar(' ');
               } /* if ELT_LEN*i */
```

```
                } /* for i */
              end2:
              q = q->Next;
              } /* while */

           /* Print the Literals exist across the table */
            i = 0;
            p = CPS.COL_Ptr;
            putchar('\n');
            Print_Line('.','.',79);
            printf("\nLiterals: ");
            while ((i<CPS.COL_Cnt) && (p!=NULL)) {
               printf("%4d",p->FAC);
               p = p->Next;
               i++;
               if ((i>0) &&
                   ((i%8)==0)) printf("\n         ");
               }
            }/* if sema */

         putchar('\n');
         Print_Line('.','.',79);
         printf("\nNumber of Columns=%3d",CPS.COL_Cnt);
         printf("\nNumber of Rows=%3d\n",CPS.ROW_Cnt);
         Print_Line('-','-',79);
         } /* Reduced Table Exists! */
    else {
       printf("\n\tSwitch Table: Completely Eliminated.");
       printf("\n\t\t\tTHE FINAL SOP CONSISTS");
       printf(" OF ALL ESSENTIAL IMPLICANTS.");
       }
}/* End Switch_Box_Report */


void Optimum_Product_Implicant_Report(Sema)
int Sema;
{
   int i;
   S_Ptr p; /* Save pointer */

   Print_Line('\n','-',79);
   printf
   ("IMPLICANTS MUST BE INCLUDED IN THE FINAL SOP:");
   p = CPS.ESS_Ptr;
   for (i=0; i<CPS.ESS_Cnt; i++) {
      if (p!=NULL) {
         printf("\n\tImplicant#:  %3d",p->IMP);
         if (p->Next!=NULL) p = p->Next;
         } /* valid pointer value */
      else printf("\n\tNULL pointer encountered !!!");
      }
```

```
   printf
      ("\n\tTotal Essential Implicants FOUND = %d\n\n",
      CPS.ESS_Cnt);
   if (Sema) {
   printf("\n.....................................");
   printf("\n:  NOTE:  THE FINAL SOP EXPRESSION    :");
   printf("\n:         MUST INCLUDES THE ABOVE %2d :");
   printf("\n:         ESSENTIAL IMPLICANTS        :",
            CPS.ESS_Cnt);
   printf("\n:.....................................:\n");
   Print_Line('#','#',79);
   putchar('\n');
   }
} /* End Optimum_Product_Implicant_Report */



void SOL_Finder_Report()
{
   int i,j,k;
   P_Ptr p;

   printf("\n\n\nSOL FINDER ELEMENTS: ");
   j = 0;

   if ((p=CPS.SOL_Ptr->SOL)!=NULL) while (p!=NULL) {
      printf("\n\tSOL#%3d: ",j++);
      for (i=0; i<LC; i++) {
         for (k=0; k<ELT_LEN; k++)
             if ((*(p->IMP_Dash+i) & Maski[k])!=0)
                printf("%3d ",In_Buf[k]);
         } /* for i */
      p = p->Next;
      } /* valid pointer value */
   else printf("\n\tNULL pointer encountered !!!");
} /* End SOL_Finder_Report */



void Absolute_Optimum_Products_Report()
{
   int i,j,k,n;
   M_Ptr p;
   E_Ptr q;

   printf("\n\nFINAL SOP:");
   if ((p = CPS.PMP_Ptr) != NULL) {
      for (i=0; i<CPS.PMP_Cnt; i++) {
         printf("\n\tSOL#%3d: Cost =%3d",i+1,p->COST);
         printf("\n\t         Implicants:  ");
```

```
         q = p->IMP_Dash;
         /* Encoded bit pattern of Implicants */
         n = 0;
         for (j=0; j<LC; j++) {
            for (k=0; k<ELT_LEN; k++) {
                if ((*(q+j) & Maski[k])!=0) {
                   printf("%5d", In_Buf[k]);
                   n++;
                   } /* if Implicant exists in the SOL */
                if (n%8==7)
                   printf("\n\t                    ");
                } /* for k */
            } /* for j */
         p = p->Next;
         } /* for i */
      printf("\n\tTotal SOL's Found = %d\n",CPS.PMP_Cnt);
      if (Execution_Option==2)
         printf("\n\nALL POSSIBLE COMBINATIONS GENERATED.");
      } /* valid pointer value */
} /* End Absolute_Optimum_Products_Report */


void CPS_Run_Options()
{
printf("\n\t##############################################");
printf("\n\t#####  COVERING PROBLEM SOLVER  (CPS)   ######");
printf("\n\t##############################################");
printf("\n\t## Phuong M. Ho              Version:  1.0 ##");
printf("\n\t## Portland State University  December 1988 ##");
printf("\n\t##------------------------------------------##");
printf("\n\t## Program Execution Options:               ##");
printf("\n\t## [0]_  Find the first N Min Cost          ##");
printf("\n\t##           solutions [default, N=1].      ##");
printf("\n\t## [1]_  Find all solutions of costs <=K.   ##");
printf("\n\t## [2]_  Find all possible solutions.       ##");
printf("\n\t##                                          ##");
printf("\n\t## Output Options:                          ##");
printf("\n\t## [0]_ Show final solutions only [default] ##");
printf("\n\t## [1]_ Show primary steps only.            ##");
printf("\n\t## [2]_ Show all works at every exec. step. ##");
printf("\n\t##     and internal structure contents.     ##");
printf("\n\t##############################################");

   printf("\n\n\t\tEnter Execution Option:\t");
   scanf("%d",&Execution_Option);
   printf("%d",Execution_Option);
   if (Execution_Option>2) Execution_Option = 0;

   if (Execution_Option==0) {
      printf("\n\t\tEnter Number of Solutions Desired:\t");
```

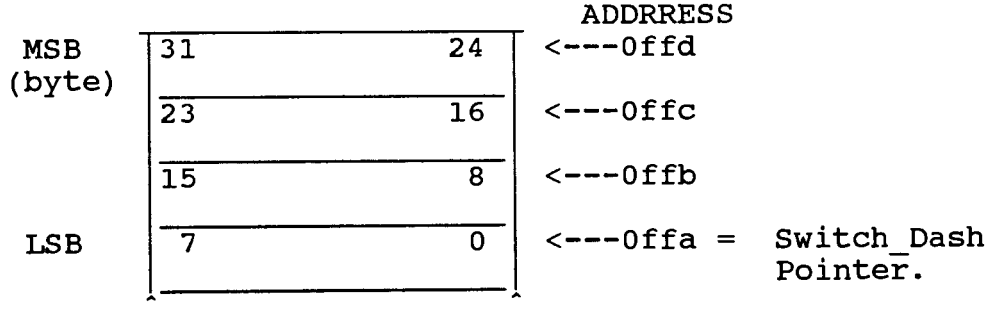
```
      scanf("%d",&NUM_SOLS_WANTED);
      printf("%d",NUM_SOLS_WANTED);
      }
   else if (Execution_Option==1) {
      printf("\n\t\tEnter Target Cost:\t");
      scanf("%d",&MAX_COST_ACCEPTABLE);
      printf("%d",MAX_COST_ACCEPTABLE);
      }

   printf("\n\t\tEnter Output Option:\t");
   scanf("%d",&Output_Option);
   if (Output_Option>2) Output_Option=0;
   printf("%d",Output_Option);

} /* End CPS_Run_Options */


/*
#########################################################
##########     M A I N       P R O G R A M     ##########
#########################################################
*/
void main()
{
   CPS_Run_Options();
   Exam_Input_Data();
   Create_CPS();
   Init_CPS();

   Get_Input_Data();
   if (Output_Option>ON) {
      Architecture_Report();
      Switch_Box_Report(3);
      }
   else if (Output_Option==ON) Switch_Box_Report(2);

start = get_time();
   Table_Reduction();
   if (Output_Option>=ON) Switch_Box_Report(2);
   Release_All_Literal_Nodes();
   Reconstruct_Implicant_Nodes();
   Create_SOL_Finder();
   SOL_Generator(Execution_Option);
   Absolute_Optimum_Products_Report();
   Optimum_Product_Implicant_Report(1);
finish = get_time();
printf("\nTOTAL EXECUTION TIME = %lf [Secs/100]\n",
                          how_long(start, finish));
}
```

```
/*-------------------------------------------------*/
/*  Program COVERING PROBLEM SOLVER (CPS)      Part#2.*/
/*  LINK LIST VERSION.                              */
/*  Table Entry & Manipulations.                    */
/*-------------------------------------------------*/
/*      Created:        MON  26 SEP 1988        17:00 */
/*      Last Edited:    THU  22 JUN 1989        18:50 */
/*-------------------------------------------------*/

#include "TABCOV.H"

/*****************************************************/
/******  E X T E R N A L   F U N C T I O N S   *******/

extern void Create_Essential_Literal_Node
                                        (Implicant Symbol);
extern void Optimum_Product_Implicant_Report(int Sema);
extern void Switch_Box_Report(int sema);
extern void Print_Line(int chr1, int chr2, int Len);

/*****************************************************/
/*****  E X T E R N A L    V A R I A B L E S   ******/

extern Header CPS;  /* Reference Node */
extern int NF;      /* Number of Literals */
extern int NL;      /* Number of Implicants */
extern int LC;      /* Length of Column strip */
extern int LR;      /* Length of Row    strip */
extern int Output_Option;/* Result-display controller */
extern Element Maski[ELT_LEN];
extern Element Masko[ELT_LEN+1];

E_Ptr Map_Row;
E_Ptr Map_Col;

/***************   P R O C E D U R E S     *************/

int Enter_Rows(Symbol,SW_Inx)
Implicant Symbol;
int SW_Inx;
{
   int i,j,k;
   L_Ptr p;

   i = 0;
   p = (L_Ptr) CPS.ROW_Ptr;
   while (p!=NULL) {
      if (p->IMP==Symbol) {
         j = SW_Inx/ELT_LEN; /* 0..15, 16..31,.. */
         k = SW_Inx%ELT_LEN;
         *(((E_Ptr) p->SW_Dash) +j) |= Maski[k];
```

```
            return(i);
            }
         else {
            p = p->Next;
            i++;
            }
         }
      return(NIL);
}


void Enter_Column(COL_Inx,SW_Inx)
int COL_Inx;/* Literal Number */
int SW_Inx; /* Row Number */
{
   int j,k;
   F_Ptr p;

   p = (F_Ptr) CPS.COL_Ptr;
   for (j=0; j<COL_Inx; j++) p = p->Next;
   p->Cnt++;
   j = SW_Inx/ELT_LEN;
   k = SW_Inx%ELT_LEN;
   *(((E_Ptr) p->SW_Dash) +j) |= Maski[k];
}



int Find_First_Bit_ON (SW_Ptr,SW_Len)
/*
    This function will search for the first bit which had
    been set "ON" in the specified switch dash. The index
    of the bit location will be returned. If no bit "ON"
    is found, the function returns NIL.
 */
E_Ptr SW_Ptr;
int SW_Len;
{
    int j,k;
    /* detect bit location in the switch dash
    */
    for (j=0; j<SW_Len; j++) {
       if (*(SW_Ptr+j) != 0x0000)
          for (k=0;k<ELT_LEN; k++)
              if (*(SW_Ptr+j)==Maski[k] )
                 return(j*ELT_LEN + k);
                        /* bit location */
       } /* for j */
    return(NIL);
} /* End Find_First_Bit_ON */
```

```
int Find_Last_Bit_ON (SW_Ptr,SW_Len)
/*
    This function will search for the last bit in the
    specified switch dash which had been set "ON". The
    index of the bit location will be returned. If
    no bit "ON" is found, the function returns NIL.
 */
E_Ptr SW_Ptr;
int SW_Len;
{
    int i,j,k;
   /* detect bit location in the switch dash
   */
    for (i=0; i<SW_Len; i++) {
       j = (SW_Len - 1) -  i;
       if (*(SW_Ptr + j) != 0x0000)
       for (k=0; k<ELT_LEN; k++) {
          if ((*(SW_Ptr+j) &
                 Maski[ELT_LEN-1-k]) != 0x0000)
           return (j*ELT_LEN + ELT_LEN-1-k);
          }  /* bit location */
       } /* for i */
    return(NIL);
} /* End Find_Last_Bit_ON */


int Remove_a_Switch(SW_Inx,Dash_Ptr,Dash_Len)
/*
```

SWITCH_DASH
MEMORY CONFIGURATION




```
    To remove a switch in the switch dash, shift the
    entire contiguous memory block (pre-allocated
    for the indicated switch dash) one bit to the
    right, start at the most significant byte (MSB)
    downto least significant byte, upto the bit
    location specificed by SW_Inx. The Least signi-
    ficant bit of the byte at higher address will be
    carried into the Most significant bit location of
    the adjacent byte at lower address. The function
```

```
      will return a non-zero value status to indicate
      that the deleted bit has been set (ie. "1"), or
      zero otherwise.
 */
int SW_Inx;
int Dash_Len;
E_Ptr Dash_Ptr; /* Address of the reduced swich dash */
{
   int i,j,k,stat;
   Element tmp; /* Temporary word,
                   used to preserve unmodified bits */

   /* compute the location indices */
   i = SW_Inx/ELT_LEN;/* Byte location */
   j = SW_Inx%ELT_LEN;/* Bit location */

   stat = *(Dash_Ptr+i) & Maski[j]; /* Status flag */

   /* current element */
   tmp = *(Dash_Ptr+i) & Masko[j];
      /* preserve unchanged bits (towards lsb) */
   *(Dash_Ptr+i) =
      ( ((*(Dash_Ptr+i) & (~Masko[j+1])) >> 1) & 0x7FFF );
                          /* shift right once, clear msb */
   *(Dash_Ptr+i) |= tmp; /* restore preserved bits */

   /* consecutive elements */
   for (k=i+1; k<Dash_Len; k++) {
      if ((*(Dash_Ptr+k) & 0x0001) != 0)
         *(Dash_Ptr+k-1) |= 0x8000;
      /* get the carry bit (lsb) of the current byte
         into the msb location of the next, adjacent byte
      */
      *(Dash_Ptr+k) = ( (*(Dash_Ptr+k) >> 1) & 0x7FFF );
                          /* shift right 1, clear msb */
      }
   return(stat);
}/* End Remove_a_Switch */



void Shrink_Row_Switch_Dash(SW_Inx)
/*
    This function will traverses through the table
    rows and reduces the length of the switch dashes,
    each by one bit at the bit location SW_Inx.
 */
int SW_Inx;
{
   int i,j;
   L_Ptr p;
```

```
   i = j = 0;
   p = (L_Ptr) CPS.ROW_Ptr;

   while ((i++< CPS.ROW_Cnt) && (p != NULL)) {
      if (Remove_a_Switch(SW_Inx,p->SW_Dash,LR) != 0)
      j++;
      p = p->Next;
      }
   if (j==0)
   printf("\n***Errors occured in Shrink Row SW_Dash.");

   if (Output_Option>ON)  {
      printf("\n\tShrink SW_Dash: Row#%d",SW_Inx);
      Switch_Box_Report(2);
      }
}/* End Shrink_Row_Switch_Dash */



void Shrink_Column_Switch_Dash(SW_Inx)
/*
    This function traverses through the table columns and
    reduces the length of the switch dashes, each by one
    bit, at the specified bit index location, SW_Inx.
 */
int SW_Inx;
{
   int i;
   F_Ptr p;

   i = 0;
   p = (F_Ptr) CPS.COL_Ptr;
   while ((i++< CPS.COL_Cnt) && (p != NULL)) {
      if (Remove_a_Switch(SW_Inx,p->SW_Dash,LC) != 0)
          p->Cnt--;
             /* Update # of bits "1" in a switch dash */
      if (p->Cnt<0) {/* might not be a negative count */
         printf("\n\tSHRINKING ERROR!!!");
         exit(88);
         }
      p = p->Next;
      }
   if (Output_Option>ON)  {
      printf("\n\tShrink SW_Dash: Column#%d",SW_Inx);
      Switch_Box_Report(1);
      }
}/* End Shrink_Column_Switch_Dash */


void Delete_A_Row_By_Index(Del_R_Inx)
/*
```

```
    This function traverses through the table rows and
    removes a Implicant node (together with its switch-
    dash) pre-indicated by the passing argument,Del_R_Inx.
 */
int Del_R_Inx;/* index to the table row to be deleted. */
{
   int i;
   L_Ptr p,q;

   if (Del_R_Inx >= CPS.ROW_Cnt)
      printf("\nError: Row# %d _ Out of table limit !!!",
         Del_R_Inx);
   else {
      if (Del_R_Inx==0) {    /* first node */
         p = CPS.ROW_Ptr;
         q = CPS.ROW_Ptr;
         CPS.ROW_Ptr = p->Next;
         }
      else { /* second to last nodes */
         p = CPS.ROW_Ptr;
         for (i=1; i<Del_R_Inx; i++)
            p = p->Next;/* Previous row */
         q = p->Next;  /* Row to be deleted */
         p->Next = q->Next;/* rerouting the pointers */
         }
      /* release the unsused memory back to the system */
      if (Output_Option>=ON)
         printf("\n\tRemoved    Row#%3d: Implicant#%4d\n",
            Del_R_Inx,q->IMP);
      free(q->SW_Dash); /* Switch dash */
      free(q);/* Implicant node */
      CPS.ROW_Cnt--;
      }/* valid index */
   if (Output_Option>ON)  Switch_Box_Report(2);
}/* End Delete_A_Row_By_Index */



int Delete_A_Row_By_Address(Del_R_Ptr)
/*
    This function will traverse through the table rows
    and will remove a Implicant node (together with its
    switch dash) pre-indicated by the passing Implicant
    pointer. If successfully performed the specified
    operations, the function will return the index of the
    deleted row (prior to deletion) to the calling routine.
    Otherwise, a NIL value will be sent back.
 */
L_Ptr Del_R_Ptr;
  /* pointer to the to_be_deleted Implicant node */
{
```

```
   int Del_R_Inx;/* index to the deleted table row. */
   L_Ptr p,q;

   Del_R_Inx = NIL; /* Initialize */

   if (Del_R_Ptr == NULL)
      printf("\nError: Invalid Implicant Pointer !!!");
   else {
      if (Output_Option>=ON)
         printf(" Implicant #%3d removed.",Del_R_Ptr->IMP);
      if ((p = CPS.ROW_Ptr) == NULL) return (NIL);
                        /* points to the first  row */
      q = p->Next;      /* points to the second row */

      if (Del_R_Ptr == p) {    /* first node */
         Del_R_Inx = 0;/* index of the node to be deleted */

         CPS.ROW_Ptr = q; /* relink the header pointer */
         }

      else { /* second to last nodes */
         Del_R_Inx = 1; /* index to the second row */
          while ((q != NULL) &&
                 (q != Del_R_Ptr) &&
                 (Del_R_Inx<CPS.ROW_Cnt)) {
            p = p->Next;  /* Previous row */
            q = p->Next;  /* Row to be deleted */
            Del_R_Inx++;
            }
         p->Next = q->Next;/* rerouting the pointer */
         }
      /* release the unsused memory back to the system */
      free(Del_R_Ptr->SW_Dash); /* Switch dash */
      free(Del_R_Ptr);/* Implicant node */
      CPS.ROW_Cnt--;
      }/* valid address */

   if (Output_Option>ON)  Switch_Box_Report(2);
   return (Del_R_Inx);
}/* End Delete_A_Row_By_Address */


void Delete_A_Column_By_Index(Del_C_Inx)
/*
    This function traverses through the table columns
    and removes a Literal node (together with its switch-
    dash) indicated by the passing argument, Del_C_Inx.
 */
int Del_C_Inx; /* index to the Column to be deleted. */
{
```

```
   int i;
   F_Ptr p,q;

   if (Del_C_Inx >= CPS.COL_Cnt)
      printf("\nError: Column#%d _ Out of table limit !",
         Del_C_Inx);
   else {
      if (Del_C_Inx==0) {    /* first node */
         p = CPS.COL_Ptr;
         q = CPS.COL_Ptr;
         CPS.COL_Ptr = p->Next;
         }
      else { /* second to last nodes */
         p = CPS.COL_Ptr;
         for (i=1; i<Del_C_Inx; i++) p = p->Next;
                        /* Previous column */
         q = p->Next;  /* Column to be deleted */
         p->Next = q->Next;/* rerouting the pointers */
         }

      /* release the unsused memory back to the system */
      if (Output_Option>=ON)
         printf("\n\tRemoved Column#%3d: Literal#%5d",
            Del_C_Inx,q->FAC);
      free(q->SW_Dash); /* Switch dash */
      free(q);/* Implicant node */
      CPS.COL_Cnt--;
      }/* valid index */

   if (Output_Option>ON)  Switch_Box_Report(1);
}/* End Delete_A_Column_By_Index */



int Delete_A_Column_By_Address(Del_C_Ptr)
/*
    This function will traverse through the table
    columns and will remove a Literal node (together
    with its switch dash) which is indicated by the
    passing Literal pointer, Del_C_Ptr.If successfully
    performed the specified operations, the function
    will return the index of the deleted column (prior
    to deletion) to the calling routine; Otherwise, a
    NIL value will be sent back.
 */
F_Ptr Del_C_Ptr;
/* pointer to the to_be_deleted Literal node */
{
   int Del_C_Inx;/* index to the deleted table column. */
   F_Ptr p,q;
```

```
   Del_C_Inx = NIL; /* Initialize */
   if (Del_C_Ptr == NULL)
      printf("\nError: Invalid Literal Pointer !!!");
   else {
      if ((p = CPS.COL_Ptr) == NULL) return (NIL);
                        /* points to the first  column */
      q = p->Next;      /* points to the second column */

      if (Del_C_Ptr == p) {    /* first node */
         Del_C_Inx = 0;/* index of the node to be deleted */

         CPS.COL_Ptr = q; /* relink the header pointer */
         }/* 1st node */
      else { /* second to last nodes */
         Del_C_Inx = 1; /* index to the second column */
          while ((q != NULL) &&
              (q != Del_C_Ptr) && (Del_C_Inx<CPS.COL_Cnt)){
            p = p->Next;  /* Previous column */
            q = p->Next;  /* Column to be deleted */
            Del_C_Inx++;
            }/* while */
         p->Next = q->Next;/* rerouting the pointer */
         }/* 2nd to last node */

      /* release the unsused memory back to the system */
      free(Del_C_Ptr->SW_Dash); /* Switch dash */
      free(Del_C_Ptr);/* Literal node */
      CPS.COL_Cnt--;
      if (Output_Option>=ON)
         printf(" Literal #%3d removed.",Del_C_Ptr->FAC);
      }/* valid address */

   if (Output_Option>ON)  Switch_Box_Report(1);
   return (Del_C_Inx);
}/* End Delete_A_Column_By_Address */




int Find_Essential_Implicants()
/*
    This function will search for an essential implicant
    (ie. a Literal containing one single Implicant). The
    function will calculate and return the corresponding
    bit set (ie. 1) location in its switch-dash. The
    returned index indicates the Implicant node to be
    removed (ie. table row). If no essential column
    detected, the function will return a NIL value. */
{
   int i,j;
   F_Ptr p;
```

```
   p = CPS.COL_Ptr;
   for (i=0; i<CPS.COL_Cnt; i++) {
      if (p->Cnt==1) {
         if ((j=Find_First_Bit_ON(p->SW_Dash,LC))!=NIL) {
            if (Output_Option>=ON)
            printf("\n\tEssential Implicants FOUND:");
            printf(" [Column#%d, Row#%d]",i,j);
            return(j); /* bit location */
            }
         } /* if only one bit set */
      p = p->Next; /* examine next Literal node */
      }
   return(NIL);
} /* End Find_Essential_Implicants */



int Remove_An_Empty_Column()
/*
    This function will search and eliminate an empty
    column, which may possibly resulted from other row/
    column reduction.  */
{
   int i;
   F_Ptr p;

   p = CPS.COL_Ptr;
   while (p!=NULL) { /* Essential Literal detection */
      if (p->Cnt==0) {
         if ((i = Delete_A_Column_By_Address(p)) != NIL)
             Shrink_Row_Switch_Dash(i);
         else printf("\n ERR in DELETE DOMINATING COLS !");
         return(i);
         }
      p = p->Next; /* examine next Literal node */
      }
   return(NIL);
} /* End Remove_An_Empty_Column */

L_Ptr Indicate_Dominated_Row(R1_Ptr,R2_Ptr)
/*
    This function will check for row domination property
    and will return the Implicant pointer to the dominated
    row (to be deleted).  If no dominated row detected,
    the function will return a NULL pointer.
 */
L_Ptr R1_Ptr,R2_Ptr; /* Pointers to 2 rows */
{
   int i;
   E_Ptr p,q;
```

```
   p = R1_Ptr->SW_Dash;
   q = R2_Ptr->SW_Dash;
   for (i=0; i<LR; i++)
       *(Map_Row+i) = *(p+i) | *(q+i);/* Bitwise OR */

   if ((i=memcmp((Element*) Map_Row,
                 (Element*) q, LR))==0) {
      if (Output_Option>=ON)
      printf
         ("\n\tImp#%3d is Dominated by Imp#%3d ***> ",
                 R1_Ptr->IMP,R2_Ptr->IMP);
      return(R1_Ptr); /* delete 1st row */
      }

   else
   if ((i=memcmp((Element*) Map_Row,
                 (Element*) p,LR))==0) {
      if (Output_Option>=ON)
      printf
         ("\n\tImp#%3d is Dominated by Imp#%3d ***> ",
                 R2_Ptr->IMP,R1_Ptr->IMP);
      return(R2_Ptr); /* delete 2nd row */
      }
   return(NULL); /* no domination, reserve both rows */
} /* End Indicate_Dominated_Row */



F_Ptr Indicate_Dominating_Column(C1_Ptr,C2_Ptr)
/*
    This function will check for column domination
    property and will return the Literal pointer to
    the dominating column (to be deleted). If no
    dominating column detected, the function will
    return a NULL pointer value.
 */
F_Ptr C1_Ptr,C2_Ptr; /* Pointers to 2 columns */
{
   int i;
   E_Ptr p,q;

   p = C1_Ptr->SW_Dash;
   q = C2_Ptr->SW_Dash;
   for (i=0; i<LC; i++)
       *(Map_Col+i) = *(p+i) | *(q+i);/* Bitwise OR */

   if ((C1_Ptr->Cnt>=C2_Ptr->Cnt) &&
       ((i=memcmp((Element*) Map_Col,
                  (Element*) p,LC))==0)) {
      if (Output_Option>=ON)  printf
         ("\n\tLit#%3d Dominates Lit#%3d **********> ",
```

```
                C1_Ptr->FAC,C2_Ptr->FAC);
        return(C1_Ptr); /* delete 1st column */
        }

   else if ((i=memcmp((Element*) Map_Col,
                      (Element*) q, LC))==0) {
      if (Output_Option>=ON) printf
         ("\n\tLit#%3d Dominates Lit#%3d ***********> ",
                C2_Ptr->FAC,C1_Ptr->FAC);
      return(C2_Ptr); /* delete 2nd column */
      }
   return(NULL); /* no domination, reserve both columns */
} /* End Indicate_Dominating_Column */



int Essential_Implicant_Remover()
/*
   ** ESSENTIAL IMPLICANTS:
   ** Detect all essential columns. Find the
   ** corresponding Implicant. Hang the corresponding
   ** Implicants to ESS_Ptr. Eliminate all other
   ** Literals which contains the preserved Implicants
   ** out of the table.
 */
{
   int i,j,k;
   L_Ptr hp;
   int Removed_Essential_Implicant;

   if (Output_Option>=ON)
      printf("\nESSENTIAL IMPLICANTS DETECTION");
   Removed_Essential_Implicant = FALSE;

   if ((CPS.ROW_Cnt==0) || (CPS.COL_Cnt==0))
      printf ("\n\tSwitch-Table Disminished !!");

   else
   while ((i = Find_Essential_Implicants())!=NIL) {
      Shrink_Column_Switch_Dash(i);
      for (hp=CPS.ROW_Ptr,j=0; j<i; j++)
         hp = hp->Next; /* ptr to i-th row */
      Create_Essential_Literal_Node(hp->IMP);
         /* save the partial product */

     /** Check the i-th row switch dash for all
         bit "ON", the columns accross these bit
         locations are containing the preserved
         Implicantls and must be removed from the
         table by rerouting the Literal pointers.
         The row switch dash will be reduced by
         the total number of bits "1".
```

```
      **/
      while ((k = Find_Last_Bit_ON
                  (hp->SW_Dash,LR)) != NIL) {
         if (Output_Option>ON)
            printf("\n\tLast Bit ON: %d",k);
         Delete_A_Column_By_Index(k);
         Shrink_Row_Switch_Dash(k);
         if ((CPS.ROW_Cnt==0) || (CPS.COL_Cnt==0))
            goto exit;
         } /* while */

      Delete_A_Row_By_Index(i); /* empty row */

      if ((CPS.ROW_Cnt>0) && (CPS.COL_Cnt>0))
         Removed_Essential_Implicant = TRUE;

      else {
         Removed_Essential_Implicant = FALSE;
         goto exit;
         } /* else */
      } /* while */

   exit:
   while (Remove_An_Empty_Column()!=NIL);
   if ((Output_Option>=ON) &&
       (Removed_Essential_Implicant))
       Switch_Box_Report(2);
   return(Removed_Essential_Implicant);
}/* End Essential_Implicant_Remover */



int Dominated_Row_Remover()
/*
   ** ROW DOMINATION:
   ** Detect all dominated rows. Remove them from the
   ** table by reconnect the Implicant pointers. Also,
   ** the corresponding columns switch dash will be
   ** reduced by invoking the Shrink_Column_Switch_Dash
   ** function to perform a right shift, overwrite the
   ** specified bit location.
 */
{

   int i,j,k;
   L_Ptr hp,tp,dp;
   int Removed_Dominated_Row;

   if (Output_Option>=ON)
      printf("\n\nROW DOMINATION DETECTION");
   Removed_Dominated_Row = FALSE;
```

```
   if (((hp = CPS.ROW_Ptr) == NULL) ||
       ((CPS.ROW_Cnt==0) ||
        (CPS.COL_Cnt==0)))
          printf ("\n\tSwitch-Table Disminished !!");

   else {
      j = 0;
      while ((j++<CPS.ROW_Cnt) &&
             (hp->Next != NULL)) {
         tp = hp->Next;  /* adjacent row */
         while (tp != NULL) {
            if ((dp = Indicate_Dominated_Row(hp,tp))
               != NULL) {
               if (dp==hp) hp = hp->Next;
               if((i=Delete_A_Row_By_Address(dp))
               != NIL)  Shrink_Column_Switch_Dash(i);
                  /* successfully deleted the specified
                     Implicant node */
               else
               printf("\n ERR while DELETE ROWS");
               if ((CPS.ROW_Cnt>0) &&
                   (CPS.COL_Cnt>0))
                  Removed_Dominated_Row = TRUE;
               else {
                  Removed_Dominated_Row = FALSE;
                  goto exit;
                  } /* must escape immediately */
               } /* a dominated row exists */
            tp = tp->Next;
            if (dp==hp) continue;
               /* skip if head pointer deleted */
            } /* while tp */
         if (dp!=hp) hp = hp->Next;
         } /* while hp */
      }/* else */
   exit:
   if ((Output_Option>=ON) &&
       (Removed_Dominated_Row))
       Switch_Box_Report(2);
   return(Removed_Dominated_Row);
}/* End Dominated_Row_Remover */




int Dominating_Column_Remover()
/*
   ** COLUMN DOMINATION:
   ** Detect all dominating columns. Remove them from
   ** the table by rerouting the Literal pntrs. Also,
   ** the corresponding row switch dash will be reduced
```

```
   ** by repeatedly calling the Remove_a_Switch function
   ** to perform a right shift, overwrite the specified
   ** bit location.
 */
{
   int i,j,k;
   F_Ptr hq,tq,dq;
   int Removed_Dominating_Column;

   if (Output_Option>=ON)
      printf ("\n\nCOLUMN DOMINATION DETECTION");
   Removed_Dominating_Column = FALSE;

   if (((hq = CPS.COL_Ptr) == NULL) ||
       ((CPS.ROW_Cnt==0) ||
        (CPS.COL_Cnt==0)))
          printf ("\n\tSwitch-Table Disminished !!");
   else {
      j = 0;
      while (Remove_An_Empty_Column()!=NIL);
      while ((j++<CPS.COL_Cnt) &&
             (hq->Next != NULL)) {
         tq = hq->Next;
            /* points to the next, adjacent column */
         while (tq != NULL) {
            if ((dq=Indicate_Dominating_Column(hq,tq))
            != NULL) {
               if (dq==hq) hq = hq->Next;
               if ((i = Delete_A_Column_By_Address(dq))
               != NIL)  Shrink_Row_Switch_Dash(i);
               else
               printf("\n ERR while DELETE COLUMNS");
               if ((CPS.ROW_Cnt>0) && (CPS.COL_Cnt>0))
                  Removed_Dominating_Column = TRUE;
               else {
                  Removed_Dominating_Column = FALSE;
                  goto exit;
                  } /* must escape immediately */
               } /* if a dominating column detected */
            tq = tq->Next;
            if (dq==hq) continue;
               /* skip the loop if head pointer deleted */
            } /* while tp */
         if (dq!=hq) hq = hq->Next;
         } /* while hp */
      } /* else */
   exit:
   if ((Output_Option>=ON) &&
       (Removed_Dominating_Column))
      Switch_Box_Report(2);
   return(Removed_Dominating_Column);
} /* End Dominating_Column_Remover */
```

```
int Table_Reduction()
{
   int i,j,k;
   int Pass;
   int Reduction_In_Progress;
   Pass = 0;
   Reduction_In_Progress = TRUE;
   while (Reduction_In_Progress) {

      if (Output_Option>=ON) {
      Print_Line('\n','#',79);
      printf("########   S W I T C H I N G   ");
      printf("T A B L E   R E D U C T I O N   ");
      printf("#######  PASS#%2d\n", ++Pass);
      Print_Line('#','#',79);
      putchar('\n');
      }
     /** ESSENTIAL IMPLICANTS: *****/
      if ((i=Essential_Implicant_Remover())==FALSE){
         if (Output_Option>=ON) printf
            ("\n\tNO Essential Implicants Detected.");
         }
     /** ROW DOMINATION: **********/
      if ((j = Dominated_Row_Remover()) == FALSE) {
         if (Output_Option>=ON)
           printf("\n\tNO Dominated Rows Detected.");
         }
     /** COLUMN DOMINATION: *******/
      if ((k = Dominating_Column_Remover())==FALSE){
         if (Output_Option>=ON) printf
         ("\n\tNO Dominating Columns Detected.");
         }

      Reduction_In_Progress =
         ((CPS.ROW_Cnt>0) &&
          (CPS.COL_Cnt>0)) ? (i | j | k) : FALSE;
      } /* while */

      if (Output_Option>=ON) {
         Print_Line('\n','#',79);
         printf("############   COMPLETE SWITCHING");
         printf(" TABLE REDUCTION PROCESS        ##");
         printf("###########\n");
         Print_Line('#','#',79);
         putchar('\n');
         Optimum_Product_Implicant_Report(0);
         }
    return(88);
} /* Table_Reduction */
```

```
/*-------------------------------------------------*/
/*  Program COVERING PROBLEM SOLVER (CPS)     Part#3.*/
/*  LINK LIST VERSION.                              */
/*-------------------------------------------------*/
/*      Created:         MON  26 SEP 1988      17:00  */
/*      Last Edited:     THU  22 JUN 1989      18:50  */
/*-------------------------------------------------*/

#include "TABCOV.H"

P_Ptr Empty_Prod;
E_Ptr Empty_Dash;
E_Ptr Cover_Dash;


/*************************************************/
/*****    E X T E R N A L   F U N C T I O N S   ****/

extern void Switch_Box_Report(int sema);
extern void SOL_Finder_Report(void);
extern void Print_Line(int chr1, int chr2, int Len);

extern int MAX_COST_ACCEPTABLE;
extern int NUM_SOLS_WANTED;



/*************************************************/
/*****   E X T E R N A L       V A R I A B L E S   **/

extern Header CPS;
extern int LC;
        /* Length of Column strip [in byte: 16 bits] */
extern int LR;
        /* Length of Row    strip [in byte: 16 bits] */
extern int Output_Option;
extern E_Ptr Map_Row; /* Working storage */

extern Implicant In_Buf[MAX_SIZ];
extern Element Maski[ELT_LEN];
extern Element Masko[ELT_LEN+1];



/*************************************************/
/**************    P R O C E D U R E S    *************/

void Release_All_Literal_Nodes ()
/*
    This function traverses through the table columns
    and removes all Lit nodes. This operation is done
    in order to miminize the working RAM space required
```

```
    to execute the program:  The process of generating
    Covering Table Solutions require no columnwise
    combination of the reduced table.
 */
{
   int i;
   F_Ptr p,q;

   if (Output_Option>ON)
      printf("\nRELEASED TABLE Literals...\n");

   if (CPS.COL_Ptr!=NULL) {
      q = CPS.COL_Ptr;
      p = q->Next;
      CPS.COL_Ptr = NULL;

      while (q!=NULL) {
      /* release the unsused memory back to the system */
         free(q->SW_Dash); /* Switch dash */
         free(q);/* Literal node */
         q = p;
         p = p->Next;
         }
      }
}/* End Release_All_Literal_Nodes */



void Reconstruct_Implicant_Nodes ()
/*
    This function traverses through the table rows and
    reduce the size of every switch dash in the Implicant
    nodes to the minimum number of chars required to
    accommodate the number of bits equal to total columns
    remains in the reduced table.
 */
{
   int i,j;
   L_Ptr p;
   E_Ptr q;


   LC = CPS.ROW_Cnt/ELT_LEN + ((CPS.ROW_Cnt%ELT_LEN)!=0);
      /* Length of new Cost (Implicant) switch dash */

   LR = CPS.COL_Cnt/ELT_LEN + ((CPS.COL_Cnt%ELT_LEN)!=0);
      /* Length of new Literal switch dash */


   /* Create an Empty Switch Dash which has all bits OFF */
```

```
    j = (LR>LC) ? LR : LC;
    Empty_Dash = (E_Ptr) calloc(j,sizeof(Element));
    for (i=0; i<j; i++) *(Empty_Dash+i) = 0x0000;

    /* Create a Cover Switch Dash which has all bits ON */
    j = CPS.COL_Cnt % ELT_LEN;
    Cover_Dash = (E_Ptr) calloc(LR,sizeof(Element));
    for (i=0; i<(LR-1); i++) *(Cover_Dash+i) = 0xFFFF;
    *(Cover_Dash+LR-1) = (j==0) ? 0xFFFF : Masko[j];


    /* Re-Initialize Input Buffer */
    for (j=0; j<MAX_SIZ; j++) In_Buf[j] =  NIL;

    j = 0;
    p = CPS.ROW_Ptr;
    while (p!=NULL) {
       q = (E_Ptr) calloc(LR,sizeof(Element));
       for (i=0; i<LR; i++) {
          *(q+i) = 0x0000;
          *(q+i) |= *((p->SW_Dash)+i);
          }
       free(p->SW_Dash); /* Switch dash */
       In_Buf[j++] = p->IMP;
       p->SW_Dash  = q;
       p = p->Next;
       }

    if (Output_Option>ON) {
       printf("\nReduced Table Implicant Switch_Dashes.\n");

       printf("\n\tSize of SW_Dashes [Words]:");
       printf(" FAC = %3d, IMP = %3d",LR,LC);
       printf("\n\tCover Dash = ");
       for (i=0;i<LR; i++)
          printf("   %4.4X", (Element) *(Cover_Dash+i));
       Switch_Box_Report(2);
       }
}/* End Reconstruct_Implicant_Nodes */


A_Ptr Create_SOL_Product_Header_Node (Prod_Cost)
/*
    Allocate contiguous memory for a SOL Header Node.
    This node contains 2 pointers: one points to next
    SOL node, the other points to a Product Node. Also,
    a counter which keeps the current cost of every
    Product node in the same level is created. All
    pointers are initialized to NULL; the counter is set
    to zero.  The function returns the starting address
    of the newly created node to the calling routine. */
```

```
int Prod_Cost;
{
     A_Ptr A;
     if((A=(A_Ptr) malloc (sizeof(SOL_Header)))==NULL) {
        printf("\nErrors in creating SOL header node.");
        exit(33);
        }

     /* Initialize the SOL Header Node */
     A->COST = Prod_Cost;
     A->SOL  = NULL;
     A->Next = NULL;
     return ((A_Ptr) A);
} /* End Create_SOL_Product_Header_Node */



P_Ptr Create_Product_Node (P_Node1, P_Node2)
/*
    Allocate memory for a Partial Product Node which
    contains a pointers pointing to next Product node
    of the same cost. A Product Node also contains 2
    pointers that point to 2 bit map patterns which
    respectively represents the existency of a Implicant
    in every Literal, and the Implicants exist in each
    partial product (ie. N Implicants ==> cost N).
 */
P_Ptr P_Node1,P_Node2;
{
     int i;
     P_Ptr PRD_Ptr;
     E_Ptr p,q;
     E_Ptr Dip_SW1, Dip_SW2;

     if ((PRD_Ptr = (P_Ptr)
        malloc (sizeof(Product_Node))) == NULL) {
        printf("\nErrors in creating product nodes.");
        exit(33);
        }

     p = (E_Ptr) calloc(LC,sizeof(Element));
        /* Dash of Implicants in the SOL */
     q = (E_Ptr) calloc(LR,sizeof(Element));
        /* Dash of Bits 1 across a row */

     /* Initialize the Product Node switch dashes */
     Dip_SW1 = P_Node1->FAC_Dash;
     Dip_SW2 = P_Node2->FAC_Dash;
     for (i=0; i<LR; i++)
        *(q+i) = *(Dip_SW1+i) | *(Dip_SW2+i);

     Dip_SW1 = P_Node1->IMP_Dash;
```

```
      Dip_SW2 = P_Node2->IMP_Dash;
      for (i=0; i<LC; i++)
         *(p+i) = *(Dip_SW1+i) | *(Dip_SW2+i);

      /* Initialize the Product Node pointers */
      PRD_Ptr->Next = NULL;
      PRD_Ptr->IMP_Dash = p;
      PRD_Ptr->FAC_Dash = q;

      if (Output_Option>ON) {
         printf("\n\tSW1=");
         for (i=0;i<LR; i++)
            printf("%4.4X",
                (Element) *(P_Node1->FAC_Dash+i));
         printf("\tSW2=");
         for (i=0;i<LR; i++)
            printf("%4.4X",
               (Element) *(P_Node2->FAC_Dash+i));
         printf("\tFAC=");
         for (i=0;i<LR; i++)
            printf("%4.4X",
               (Element) *(PRD_Ptr->FAC_Dash+i));
         }
      return ((P_Ptr) PRD_Ptr);
} /* End Create_Product_Node */



void Release_Product_Node (Node_Ptr)
/*
    De-Allocate memory for a Partial Product Node which
    has been described in detailed in procedure Create_
    Product_Node.
 */

P_Ptr Node_Ptr;
{
      free( (E_Ptr) Node_Ptr->FAC_Dash);
      free( (E_Ptr) Node_Ptr->IMP_Dash);
      free( (P_Ptr) Node_Ptr);

} /* End Release_Product_Node */



void Create_PMP_Node (Prod_Cost, Min_Prod_Ptr)
/*
    Allocate memory for a Partial Resultant Product
    Node which contains a pointers pointing to next
    Minimum Product node, a counter which keeps the
    Product cost, and a bit map representation of the
```

```
    Implicants appeared in the resultant product. These
    node will be connected to the CPS structure
    linked-list. The final results would be the
    concatenation of these nodes' Implicant strips
    with that of Essential Implicant nodes.
 */
int Prod_Cost;
P_Ptr Min_Prod_Ptr;
{
     int i;
     M_Ptr h,p;
     E_Ptr q;

    /* Create a new Minimum Product node */
     p = (M_Ptr) malloc (sizeof(Minimum_Product));
     q = (E_Ptr) calloc (LC,sizeof(Element));
        /* Implicant_Dash in PMP Node */
     for (i=0; i<LC; i++)
        *(q+i) = *((Min_Prod_Ptr->IMP_Dash) +i);
     p->COST = Prod_Cost;
     p->IMP_Dash = q;
     p->Next  = NULL;

    /* Link the PMP node to the CPS linked list */
     if (CPS.PMP_Cnt==0) CPS.PMP_Ptr = p;
     else {
        h = (M_Ptr) CPS.PMP_Ptr; /* Start Pointer */
        for (i=1; i<CPS.PMP_Cnt; i++)
           h = h->Next; /* Traverse to eolist */
        h->Next = p; /* link it in */
        }
     CPS.PMP_Cnt++;
     Release_Product_Node(Min_Prod_Ptr);
        /* Free the given Product node */
     if (Output_Option>=ON)
         printf("\tPMP Node#%3d is CREATED.\n",
            CPS.PMP_Cnt);

} /* End Create_PMP_Node */



void Create_SOL_Finder ()
/*
    Creates the initial SOL finder network, which
    contains a Product-Header node connected to the
    CPS structure. All product nodes of cost 1 will be
    created by this function. The first product node will
    be linked to the Product-Header node, which  is pointed
    to by SOL_Ptr, pointer element of the CPS structure. */
{
```

```
      int i,j;
      L_Ptr h;
      P_Ptr p,q;

      /* Create an Empty Product Node */
      Empty_Prod = (P_Ptr) malloc (sizeof(Product_Node));
      Empty_Prod->Next = NULL;
      Empty_Prod->IMP_Dash =
      Empty_Prod->FAC_Dash = Empty_Dash;

      /* Create first Header node which points to
         products of cost 1
      */
      CPS.SOL_Ptr = (A_Ptr) Create_SOL_Product_Header_Node(1);

      j = 0;
      h = CPS.ROW_Ptr; /* Address of the 1st Implicant node */
      while (h!=NULL) {
         q = (P_Ptr) Create_Product_Node
                          (Empty_Prod,Empty_Prod);
         for (i=0; i<LR; i++)
            *((q->FAC_Dash)+i) |= *(h->SW_Dash +i);
         *((q->IMP_Dash)+(j/ELT_LEN)) |= Maski[j%ELT_LEN];
         if (j++==0) CPS.SOL_Ptr->SOL = p = q;
         else {
            p->Next = q;
            p = p->Next;
            }
         h = h->Next;
         }
      if (Output_Option>ON) {
         printf("\n\tSOL Finder Created...");
         SOL_Finder_Report();
         }

} /* End Create_SOL_Finder */



A_Ptr Start_New_Finder_Level
                (Prod_Cost, Prod1_P, Prod2_P)
/*
    Creates a new Product header node of cost i = j+1 at
    the succeeding level by combining two product nodes
    of cost j in the present level. This function is
    invoked as the pointer to the first product node is
    advanced along the linked list.
 */
int Prod_Cost;
P_Ptr Prod1_P, Prod2_P;
{
```

```
      A_Ptr h;

      if (Output_Option>ON)
         printf("\nA New Finder Level started.");

      /* Create a Product Header node..
         define a new level.
      */
      h = (A_Ptr)
          Create_SOL_Product_Header_Node(Prod_Cost);
      /* Create the 1st Product node in this newly
         created level
      */
      h->SOL = (P_Ptr)
               Create_Product_Node(Prod1_P,Prod2_P);
      return(h);
} /* End Start_New_Finder_Level */



void Remove_Previous_Finder_Level (Header_Ptr)
/*
    Remove the last Product nodes level. Reroute
    the CPS->SOL_Ptr element to the most current
    finder level. This function is executed for
    saving the located system memory during program
    execution only; therefore, it could be ignored
    in case of the execution speed is more desirable
 */
A_Ptr Header_Ptr;
{
      int j,k;
      A_Ptr h;
      P_Ptr p,q;

      /* Rerouting CPS pointer element, SOL_Ptr */
      h = (A_Ptr) Header_Ptr;
      CPS.SOL_Ptr = (A_Ptr) Header_Ptr->Next;

      /* Remove all Product Nodes attached in the
         indicated finder level
      */
      p = (P_Ptr) h->SOL;
      while (p!=NULL) {
         if (Output_Option>ON) {
             printf("\n\tNodes Removed: FAC= ");
             for (j=0;j<LR; j++) printf
                ("%4.4X", (Element) *(p->FAC_Dash+j));
             printf("\tIMP= ");
             for (j=0; j<LC; j++) {
                for (k=0; k<ELT_LEN; k++) {
```

```
                    if ( (*(p->IMP_Dash+j) & Maski[k]) !=0)
                       printf("%3d ",In_Buf[k]);
                    } /* for k */
                 } /* for j */
              } /* Debug option ON */
        q = p;
        p = p->Next;
        Release_Product_Node(q); /* Free a Product node */
        }
     free ( (A_Ptr) h);
        /* Release the Product Header node */
     if (Output_Option>ON)
        printf("\nPrevious Finder Level Removed.");

} /* End Remove_Previous_Finder_Level */


int Product_Node_Domination_Check (Dash1_Ptr, Dash2_Ptr)
/*
    Check the domination property of a pair of product
    nodes. An integer returned code will indicate the
    detected domination property:
         0 ... No domination (2 nodes are different).
         1 ... Node #1 dominates Node #2.
         2 ... Node #2 dominates Node #1.
 */
E_Ptr Dash1_Ptr, Dash2_Ptr;
{
     int i; /* Temp variable */
     for (i=0; i<LR; i++)
        *(Map_Row+i) = *(Dash1_Ptr+i) | *(Dash2_Ptr+i);

     if ((i=memcmp((Element*) Map_Row,
                   (Element*) Dash1_Ptr, LR))==0)
        return(1);
        /* Dash1_Ptr dominates Dash2_Ptr:
           Node #1 dominates Node #2.
        */
     else if ((i=memcmp((Element*) Map_Row,
                        (Element*) Dash2_Ptr,LR))==0)
        return(2);
        /* Dash2_Ptr dominates Dash1_Ptr:
           Node #2 dominates Node #1.
         */
     return(0);  /* No domination. */

} /* Product_Node_Domination_Check */


int Linking_New_Product_Node (Head_Ptr, Node_Ptr)
```

```
/*
    Check the domination property of the new product
    node against the existing nodes of the same cost
    in the given level (where start pointer is Head_Ptr).
    Delete the product node which has dominated Literal
    Switch Dash.
    Returns the status of the operation:
      0:  No link is made (ie. New node is ignored).
      1:  New Node is linked in. Existing nodes are
          unaffected.
      2:  New Node is linked in. At least an existing
          node is removed.
 */
A_Ptr Head_Ptr;
P_Ptr Node_Ptr;
{
     int i;
     Boolean node1_removed;
       /* flag to indicate if the 1st node is deleted */
     P_Ptr h,t,r;
     E_Ptr p,q;

     node1_removed = TRUE;
     h = t = r = Head_Ptr->SOL;
        /* Pntrs to 1st Prod Node in current level */
     q = (E_Ptr) Node_Ptr->FAC_Dash;
        /* New Node. Exammined to be entered */

     if (h!=NULL) while (h!=NULL) {
        p = (E_Ptr) h->FAC_Dash;
             /* Existing Node(s) in the Linked-List */
        i = Product_Node_Domination_Check(p,q);
        if (i==2) {
           /* q dominates p: New node dominates Existing
              node. Remove the Old node. Reroute the
              pointers.  If aproached the end of list
              then Attach the New node in.  Otherwise,
              advance the pointer and try another node.
           */
           if (node1_removed = TRUE) {
              /* Remove the 1st node of the list */
              if (h->Next==NULL) {
                 /* Approached the Last Node */
                 Head_Ptr->SOL = Node_Ptr;
                    /* Link the New node in */
                 Release_Product_Node(t);
                    /* Release the existing node */
                 return(2); /* 1st node is dominated */
                 }
              else { /* Not the End of the linked-list */
                 r = h = h->Next;
                    /* Advance h pointer to next node */
```

```
            Release_Product_Node(t);
               /* Release the existing node */
            t = h;
               /* Pointer to the next product node
                  in the same level */
            }
         } /* Removed node is the first node in
              the current level */

   /* Remove existing node in the middle
      of the list */
    else {
       if (h->Next==NULL) {
         /* Approached the Last Node */
          Release_Product_Node(h);
             /* Release the existing node */
          t->Next = Node_Ptr;
             /* Reroute, link new node in */
          Head_Ptr->SOL = r;
             /* Reassign pointer to new 1st node */
          return(2); /* 1st node is dominated */
          }
       else { /* Not the End node of the list */
          h = h->Next;
             /* Advance h pointer to next node */
          Release_Product_Node(t->Next);
             /* Release the existing node */
          t->Next = h;
             /* Point to next node in the level */
          }
       } /* Removed node is not the first node
            in the current level */
    } /* Existing node is Dominated */


else if (i==1) {
   /* p dominates q: Old node dominates New node.
      Ignore the new node. Do nothing.
   */
   Release_Product_Node(Node_Ptr);
      /* Release the New node */
   Head_Ptr->SOL = r;
      /* Reassign pointer to new 1st node */
   return(0); /* New node is dominated by 1st node */

   } /* New node is Dominated */

else if (i==0) {
   /* No domination detected.
      If approach the end of the list, link the
      new node in. */
   if (h->Next==NULL) {
```

```
                h->Next = Node_Ptr; /* Last node */
                Head_Ptr->SOL = r;
                   /* Reassign pointer to new 1st node */
                return(1);
                }
             else {
             /* Reset the Flag; Let the t pointer trails
                the h pointer by one node. This is necessary
                for removing node in the middle or at the
                end of the linked-list.
             */
                if (node1_removed==TRUE)
                   node1_removed = FALSE;
                else t = t->Next;
                   /* Advance t pointer to next node */
                h = h->Next;
                   /* Advance h pointer to next node */
                }
             }/* No Domination */
          }/* while traversing */

      else { /* Link the new Node in */
         Head_Ptr->SOL = Node_Ptr;
            /* Link the New node in */
         return(0); /* 1st node is the New node */
         }

} /* End Linking_New_Product_Node */



Boolean An_SOL_is_Found(Prod_Ptr)
/*
    This function is invoked at any time a new Product
    node is created. The resultant codes returned are:
       0 : The given node does not satisfy the given
           SOP expression.
       1 : The given node is a Partial Minimum Product.
 */
P_Ptr Prod_Ptr;
{
    int i;
    /* memcmp ==> <0: buf1<buf2;
                  0: buf1=buf2;  >0: buf1>buf2.
    */
    if ((i= memcmp((Element*) Cover_Dash,
                   (Element*) Prod_Ptr->FAC_Dash,
                    LR) )==0) {
       if (Output_Option>=ON)
          printf("\nAn SOL has been FOUND! ===>");
       return(TRUE);
       }
```

```
    return(FALSE);
} /* End An_SOL_is_Found */


int SOL_Generator (Option)
/*
    Generates potential Absolute Optimized Products by
    combining pairs of products nodes, each of cost j,
    to form a new product node of cost j+1. This function
    is executed repetetively until at least one of these
    conditions has been met:
       0)_ Generating the first N SOL Products of
           smallest possible costs.
           Returned Code: 0.
       1)_ Satisfying a user specification (eg. find
           only optimized products of cost k).
           Returned Code: 1.
       2)_ Generating all SOLs (ie. exhaustedly
           generating all possible combinations).
           Returned Code: 2.
 */
int Option;
{
     A_Ptr h,t,r;
     P_Ptr p,q,v;
     Boolean new_level;

     Print_Line('\n','#',79);
     printf("#########   ABSOLUTE OPTIMIZED PRODUCT GEN");
     printf("ERATOR IN OPERATION    ############\n");
     Print_Line('#','#',79);
     putchar('\n');
     h = t = CPS.SOL_Ptr; /*  1st header node */
     if ((q = h->SOL->Next)!= NULL) {
        while (h!=NULL) {
           if (Output_Option>ON)
              printf("\n\tIn h_loop.");
           p = h->SOL;
             /* 1st node in the current level */
           while (p!=NULL) {
              if (Output_Option>ON)
                 printf("\n\tIn p_loop.");
              new_level = TRUE;
              q = p->Next; /* 2nd node in current level */

              while (q!=NULL) {
                 if (new_level == TRUE) {
                    /* Start a new level product nodes */
                    t->Next =
                      Start_New_Finder_Level(h->COST+1,p,q);
```

```
            t = t->Next;
            /* Point to header node in next level */
            v = t->SOL;
            }
         else
         /* Generate more nodes in the same level */
            v = (P_Ptr) Create_Product_Node(p,q);

         if (Output_Option>ON)
            printf("\t....In q_loop.");

         if (An_SOL_is_Found(v)) {
         /* Absolute Optimum Product */
            if (new_level==TRUE) { /* is found */
               t->SOL = NULL;
               new_level = FALSE;
               }
            Create_PMP_Node (h->COST+1, v);
               /* Resultant linked-list */
            if (Option==0) {
               /* Find 1st N min cost products */
               if (CPS.PMP_Cnt>=NUM_SOLS_WANTED)
                  return(0);
               }
            }/* FOUND AN OPTIMUM PRODUCT */
         else {
            if (new_level==TRUE)
               new_level = FALSE;
            else Linking_New_Product_Node(t,v);
            }/* Product Node is not and SOL */

         if (Option==1) {
            /* Products of cost less than K only */

            if (((h->COST)+1+CPS.ESS_Cnt)>
                  MAX_COST_ACCEPTABLE) {
               if (CPS.PMP_Cnt==0)
                 printf
                    ("\n\tNO SOLs of cost <= %3d.",
                 MAX_COST_ACCEPTABLE);
               else printf
                 ("\n\tNO other SOLs of cost<=%3d.",
                 MAX_COST_ACCEPTABLE);
               return(1);
               }
            }
         q = q->Next;
         } /* while q */
      p = p->Next;
      } /* while p */

   r = h; /* Pointer to removed header node */
```

```
            h = h->Next;
            Remove_Previous_Finder_Level(r);
            } /* while h */
          } /* if more than 1 nodes exist in first level */

      else printf("\nCPS System: No Reduced Table !!");
      return(2);
} /* End SOL_Generator */
```

```
/*-------------------------------------------------------*/
/* Program COVERING PROBLEM SOLVER:       Include File. */
/*                                                       */
/* This include file contains type definitions,constant */
/* definitions,and external variable declarations to be */
/* included into modules of the Covering Problem Solver */
/* program. The definitions of these variables could be */
/* found in the main module, Part#1, File TABCOV.C      */
/*-------------------------------------------------------*/
/*     Created:         WED  26 OCT 1988        12:00    */
/*     last Edited:     FRI  16 DEC 1988        23:00    */
/*-------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>

/*****   C O N S T A N T S   D E F I N I T I O N S   ***/

#define MAX_SIZ 1024
#define NIL     9999
#define ELT_LEN 16
#define OFF     0
#define ON      1
#define FALSE   0
#define TRUE    1

/*****    T Y P E     D E F I N I T I O N S     ******/

typedef char Boolean;
typedef int Implicant;
typedef unsigned Element;
typedef Element *E_Ptr;

typedef struct Tbl_Col {
      int Cnt;
      Implicant FAC;
      E_Ptr SW_Dash;
      struct Tbl_Col *Next;
      } Literal_Strip;
typedef Literal_Strip  *F_Ptr;

typedef struct Tbl_Row {
      Implicant IMP;
      E_Ptr SW_Dash;
      struct Tbl_Row *Next;
      } Implicant_Strip;
typedef Implicant_Strip *L_Ptr;

typedef struct Ess_Fac {
      Implicant IMP;
```

```
      struct Ess_Fac *Next;
      } Essential_Literal;
typedef Essential_Literal *S_Ptr; /* Save pointer */

typedef struct Min_Prod {
      int COST;
      E_Ptr IMP_Dash;
      struct Min_Prod *Next;
      } Minimum_Product;
typedef Minimum_Product *M_Ptr;

typedef struct Prod {
      E_Ptr IMP_Dash;
      E_Ptr FAC_Dash;
      struct Prod *Next; /* Neighbor node, same cost */
      } Product_Node;
typedef Product_Node *P_Ptr;

typedef struct Prod_Head {
      int COST;
      P_Ptr SOL;
      struct Prod_Head *Next;
      } SOL_Header;
typedef SOL_Header *A_Ptr;

typedef struct Header_Node {
      int ESS_Cnt;
      int PMP_Cnt;
      int COL_Cnt;
      int ROW_Cnt;
      S_Ptr ESS_Ptr;
      M_Ptr PMP_Ptr;
      A_Ptr SOL_Ptr;
      F_Ptr COL_Ptr;
      L_Ptr ROW_Ptr;
      } Header;  /* Data structure of a header node */
```

```
/*
##################################################
#    COVERING PROBLEM GENERATOR                  #
#------------------------------------------------#
#    Created:       TUE 15 NOV 88      15:00     #
#    Last Edited:   THU 17 NOV 88      14:00     #
##################################################
*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void srand(seed);
     unsigned seed;
int rand(void);

rand()
{
   static int randx = (unsigned) seed_rand;
   randx = (randx * 25173 +13849) % 65536;
   return (randx);
}

main(a1,a2)
int a1;
char *a2[];
{
   static int i, j, k, m, n, p, y, z;
   unsigned seed_rand;
   int buf[111];
   FILE *fp;

   printf("\n GENERATE DATAFILE: %12s",a2[1]);
   if ((fp = fopen(a2[1],"w")) == (FILE *) NULL)
      printf("\n Sorry, cannot create file '%12s'!\n",
             a2[1]);
   else {
      printf("\n Number of Implicants  [ROWS]:\t");
      scanf("%d", &i);
      printf("\n Number of Literals [COLUMNS]:\t");
      scanf("%d", &j);
      printf("\n MaxNum of Implicants/Literals:\t");
      scanf("%d", &y);
      printf("\n Seed for random sequence[int]:\t");
      scanf("%d", &seed_rand);

      if ((i<=0) || (i>256))  i=8; /* Default */
      if ((j<=0) || (j>1000)) j=8;
      if ((y<=0) || (y>100))  j=8;

      fprintf(fp,"\n\t");
```

```
      srand(seed_rand);
      for (m=0; m<j; m++) { /* Literals: COL */

         z = rand() % y;
         z = (z>=0)? (z+1) : -z;
         if (z>1) /* number of Imp/Lit */
            for (n=0; n<z; n++) {

            p = rand() % i;
            p = (p>=0)? (p+1) : -p;

            /* Reject redundant Literals in
               the same Implicant
            */
            if (n==0)
               buf[0] = p;
            else
            for (k=0; k<n; k++)
               if (p==buf[k]) p=0;
            if (p==0)
               n--;
            else buf[n] = p;
            } /* for n */

         /* write the buffer to file
         */
         for (n=0, k=1; n<z; n++, k++) {
            printf("\t%d",buf[n]);
            fprintf(fp, "%4d", buf[n]);
            if (k>=16) {
               printf("\n");
               fprintf(fp,"\n\t");
               k = 0;
               }
            } /* for n */

         printf ("\n9999\n");
         fprintf (fp, "\n\t9999\n\t");
         } /* for m */
      }
   fclose(fp);
} /* End MAIN */
```

APPENDIX E

THE CPS ALGORITHM - TEST CASES AND RESULTS

```
################################################
#######  COVERING PROBLEM SOLVER  (CPS)   #######
################################################
## Phuong M. Ho                    Version:  1.0  ##
## Portland State University    December 1988  ##
##--------------------------------------------##
## Program Execution Options:                 ##
##   [0]_  Find the first N Min Cost solutions.##
##                                [default, N=1].##
##   [1]_  Find all solutions of costs <=K.    ##
##   [2]_  Find all possible solutions.        ##
##                                             ##
## Output Options:                            ##
##   [0]_ Show final solutions only [default]. ##
##   [1]_ Show primary steps only.             ##
##   [2]_ Show all works at every exec. step.  ##
##        and internal structure contents.     ##
################################################


     Enter Execution Option:  0
     Enter Number of Solutions Desired: 1
     Enter Output Option:     1
     Enter Name of Data File: INFILE.DAT


#########################################################
LIST OF IMPLICANTS:
        1     2     3     4     5     6     7     8
        9    10    11    12    13    14    15    16
       17    18    19    20    21    22    23    24
       25
    Number of Implicants =    25
    Number of Literals       =    25
    Size of Switch_Dash [in 16-bit-word unit]
         Implicant Switch_Dash:   2 words.
         Literal   Switch_Dash:   2 words.

---------------------------------------------------------
               S W I T C H     T A B L E
               Implicants versus Literals
---------------------------------------------------------
Implicant#   1:    00001000 00101000 00000100 0
Implicant#   2:    00111011 10000001 10000111 0
Implicant#   3:    01000110 00000101 11000000 0
Implicant#   4:    00001100 00000000 10010111 0
Implicant#   5:    00000000 00010001 01000000 0
```

```
Implicant#   6:    00100000 10011001 00110001 0
Implicant#   7:    00000000 10000010 11110011 0
Implicant#   8:    00000100 10000010 01010000 0
Implicant#   9:    00001000 10000000 11001001 0
Implicant#  10:    01000101 00010001 10111101 0
Implicant#  11:    00100111 01010001 00100000 1
Implicant#  12:    00000100 00000010 00101101 1
Implicant#  13:    00100000 00100001 10110000 0
Implicant#  14:    00100100 10000001 01001001 1
Implicant#  15:    00100000 10111101 11101000 0
Implicant#  16:    10101010 00110001 01110100 0
Implicant#  17:    01001100 00101000 11010000 0
Implicant#  18:    00100010 00000000 11001000 0
Implicant#  19:    01000000 00000100 00100000 0
Implicant#  20:    01000001 01101000 01110110 0
Implicant#  21:    00000100 00100000 11011101 1
Implicant#  22:    01000000 11000001 10100000 0
Implicant#  23:    10000100 00100001 00011101 0
Implicant#  24:    00001000 00100001 10111001 1
Implicant#  25:    00001001 01010010 11011100 0
.............................................................
Literals:   0   1   2   3   4   5   6   7
            8   9  10  11  12  13  14  15
           16  17  18  19  20  21  22  23
           24
.............................................................
Number of Columns= 25
Number of Rows= 25
-------------------------------------------------------------


#############################################################
        S W I T C H    T A B L E   R E D U C T I O N
        PASS# 1
#############################################################

ESSENTIAL IMPLICANTS DETECTION
     Essential Implicants FOUND: [Column#3, Row#1]
     Removed Column# 23: Literal#   23
     Removed Column# 22: Literal#   22
     Removed Column# 21: Literal#   21
     Removed Column# 16: Literal#   16
     Removed Column# 15: Literal#   15
     Removed Column#  8: Literal#    8
     Removed Column#  7: Literal#    7
     Removed Column#  6: Literal#    6
     Removed Column#  4: Literal#    4
     Removed Column#  3: Literal#    3
     Removed Column#  2: Literal#    2
     Removed    Row#  1: Implicant#  2
```

```
-----------------------------------------------------------
                S W I T C H     T A B L E
                Implicants versus Literals
-----------------------------------------------------------
Implicant#    1:     00001010 000000
Implicant#    3:     01100001 010000
Implicant#    4:     00100000 000100
Implicant#    5:     00000100 010000
Implicant#    6:     00000110 001100
Implicant#    7:     00000000 111100
Implicant#    8:     00100000 110100
Implicant#    9:     00000000 010010
Implicant#   10:     01100100 001110
Implicant#   11:     00110100 001001
Implicant#   12:     00100000 101011
Implicant#   13:     00001000 001100
Implicant#   14:     00100000 010011
Implicant#   15:     00001111 011010
Implicant#   16:     10001100 011100
Implicant#   17:     01101010 010100
Implicant#   18:     00000000 010010
Implicant#   19:     01000001 001000
Implicant#   20:     01011010 011100
Implicant#   21:     00101000 010111
Implicant#   22:     01010000 001000
Implicant#   23:     10101000 000110
Implicant#   24:     00001000 001111
Implicant#   25:     00010100 110110
.............................................................
Literals:    0   1   5   9  10  11  12  13
            14  17  18  19  20  24
.............................................................
Number of Columns= 14
Number of Rows= 24
-----------------------------------------------------------

ROW DOMINATION DETECTION
Imp#  1 is Dominated by Imp# 15 ***>  Implicant #  1 removed.
Imp#  4 is Dominated by Imp#  8 ***>  Implicant #  4 removed.
Imp#  5 is Dominated by Imp# 15 ***>  Implicant #  5 removed.
Imp#  9 is Dominated by Imp# 14 ***>  Implicant #  9 removed.
Imp# 13 is Dominated by Imp# 16 ***>  Implicant # 13 removed.
Imp# 18 is Dominated by Imp# 14 ***>  Implicant # 18 removed.
Imp# 14 is Dominated by Imp# 21 ***>  Implicant # 14 removed.
Imp# 22 is Dominated by Imp# 20 ***>  Implicant # 22 removed.

-----------------------------------------------------------
                S W I T C H     T A B L E
                Implicants versus Literals
-----------------------------------------------------------
Implicant#    3:     01100001 010000
Implicant#    6:     00000110 001100
```

```
Implicant#   7:    00000000 111100
Implicant#   8:    00100000 110100
Implicant#  10:    01100100 001110
Implicant#  11:    00110100 001001
Implicant#  12:    00100000 101011
Implicant#  15:    00001111 011010
Implicant#  16:    10001100 011100
Implicant#  17:    01101010 010100
Implicant#  19:    01000001 001000
Implicant#  20:    01011010 011100
Implicant#  21:    00101000 010111
Implicant#  23:    10101000 000110
Implicant#  24:    00001000 001111
Implicant#  25:    00010100 110110
.............................................................
Literals:   0   1   5   9  10  11  12  13
           14  17  18  19  20  24
.............................................................
Number of Columns= 14
Number of Rows= 16
-------------------------------------------------------------

COLUMN DOMINATION DETECTION
Lit# 10 Dominates Lit#  0 ***********>  Literal # 10 removed.
Lit# 19 Dominates Lit#  0 ***********>  Literal # 19 removed.

-------------------------------------------------------------
                S W I T C H     T A B L E
                Implicants versus Literals
-------------------------------------------------------------
Implicant#   3:    01100010 1000
Implicant#   6:    00001100 0100
Implicant#   7:    00000001 1100
Implicant#   8:    00100001 1000
Implicant#  10:    01101000 0110
Implicant#  11:    00111000 0101
Implicant#  12:    00100001 0111
Implicant#  15:    00001110 1110
Implicant#  16:    10001000 1100
Implicant#  17:    01100100 1000
Implicant#  19:    01000010 0100
Implicant#  20:    01010100 1100
Implicant#  21:    00100000 1011
Implicant#  23:    10100000 0010
Implicant#  24:    00000000 0111
Implicant#  25:    00011001 1010
.............................................................
Literals:   0   1   5   9  11  12  13  14
           17  18  20  24
.............................................................
Number of Columns= 12
Number of Rows= 16
```

```
##########################################################
        S W I T C H    T A B L E    R E D U C T I O N
        PASS# 2
##########################################################

ESSENTIAL IMPLICANTS DETECTION
NO Essential Implicants Detected.

ROW DOMINATION DETECTION
Imp#  6 is Dominated by Imp# 15 ***>  Implicant #  6 removed.
Imp# 24 is Dominated by Imp# 12 ***>  Implicant # 24 removed.

----------------------------------------------------------
                S W I T C H     T A B L E
                Implicants versus Literals
----------------------------------------------------------
Implicant#   3:    01100010 1000
Implicant#   7:    00000001 1100
Implicant#   8:    00100001 1000
Implicant#  10:    01101000 0110
Implicant#  11:    00111000 0101
Implicant#  12:    00100001 0111
Implicant#  15:    00001110 1110
Implicant#  16:    10001000 1100
Implicant#  17:    01100100 1000
Implicant#  19:    01000010 0100
Implicant#  20:    01010100 1100
Implicant#  21:    00100000 1011
Implicant#  23:    10100000 0010
Implicant#  25:    00011001 1010
..........................................................
Literals:    0   1   5   9  11  12  13  14
            17  18  20  24
..........................................................
Number of Columns= 12
Number of Rows= 14
----------------------------------------------------------
COLUMN DOMINATION DETECTION
Lit#  5 Dominates Lit# 24 ***********>  Literal #  5 removed.
Lit# 17 Dominates Lit# 12 ***********>  Literal # 17 removed.

----------------------------------------------------------
                S W I T C H     T A B L E
                Implicants versus Literals
----------------------------------------------------------
Implicant#   3:    01000100 00
Implicant#   7:    00000011 00
Implicant#   8:    00000010 00
Implicant#  10:    01010001 10
Implicant#  11:    00110001 01
```

```
Implicant#  12:    00000011 11
Implicant#  15:    00011101 10
Implicant#  16:    10010001 00
Implicant#  17:    01001000 00
Implicant#  19:    01000101 00
Implicant#  20:    01101001 00
Implicant#  21:    00000000 11
Implicant#  23:    10000000 10
Implicant#  25:    00110010 10
.....................................................
Literals:   0   1   9  11  12  13  14  18
           20  24
.....................................................
Number of Columns= 10
Number of Rows= 14



#########################################################
        S W I T C H   T A B L E   R E D U C T I O N
        PASS# 3
#########################################################

ESSENTIAL IMPLICANTS DETECTION
NO Essential Implicants Detected.

ROW DOMINATION DETECTION
Imp#  3 is Dominated by Imp# 19 ***>  Implicant #  3 removed.
Imp#  8 is Dominated by Imp# 12 ***>  Implicant #  8 removed.
Imp# 21 is Dominated by Imp# 12 ***>  Implicant # 21 removed.
Imp# 17 is Dominated by Imp# 20 ***>  Implicant # 17 removed.


-------------------------------------------------------
               S W I T C H    T A B L E
               Implicants versus Literals
-------------------------------------------------------
Implicant#   7:    00000011 00
Implicant#  10:    01010001 10
Implicant#  11:    00110001 01
Implicant#  12:    00000011 11
Implicant#  15:    00011101 10
Implicant#  16:    10010001 00
Implicant#  19:    01000101 00
Implicant#  20:    01101001 00
Implicant#  23:    10000000 10
Implicant#  25:    00110010 10
.....................................................
Literals:   0   1   9  11  12  13  14  18
           20  24
.....................................................
```

```
Number of Columns= 10
Number of Rows= 10
-----------------------------------------------------------
COLUMN DOMINATION DETECTION
Lit# 18 Dominates Lit#  1 ***********>  Literal # 18 removed.

-----------------------------------------------------------
                S W I T C H     T A B L E
                Implicants versus Literals
-----------------------------------------------------------
Implicant#   7:    00000010 0
Implicant#  10:    01010001 0
Implicant#  11:    00110000 1
Implicant#  12:    00000011 1
Implicant#  15:    00011101 0
Implicant#  16:    10010000 0
Implicant#  19:    01000100 0
Implicant#  20:    01101000 0
Implicant#  23:    10000001 0
Implicant#  25:    00110011 0
...........................................................
Literals:   0   1   9  11  12  13  14  20  24
...........................................................
Number of Columns=  9
Number of Rows= 10

###########################################################
         S W I T C H    T A B L E   R E D U C T I O N
         PASS# 4
###########################################################

ESSENTIAL IMPLICANTS DETECTION
NO Essential Implicants Detected.

ROW DOMINATION DETECTION
Imp#  7 is Dominated by Imp# 12 ***>  Implicant #  7 removed.
-----------------------------------------------------------
                S W I T C H     T A B L E
                Implicants versus Literals
-----------------------------------------------------------
Implicant#  10:    01010001 0
Implicant#  11:    00110000 1
Implicant#  12:    00000011 1
Implicant#  15:    00011101 0
Implicant#  16:    10010000 0
Implicant#  19:    01000100 0
Implicant#  20:    01101000 0
Implicant#  23:    10000001 0
Implicant#  25:    00110011 0
...........................................................
Literals:   0   1   9  11  12  13  14  20  24
...........................................................
```

```
Number of Columns=  9
Number of Rows=  9
------------------------------------------------------------
COLUMN DOMINATION DETECTION
Lit# 20 Dominates Lit# 14 ***********>  Literal # 20 removed.


------------------------------------------------------------
                S W I T C H     T A B L E
                Implicants versus Literals
------------------------------------------------------------
Implicant#  10:    01010000
Implicant#  11:    00110001
Implicant#  12:    00000011
Implicant#  15:    00011100
Implicant#  16:    10010000
Implicant#  19:    01000100
Implicant#  20:    01101000
Implicant#  23:    10000000
Implicant#  25:    00110010
............................................................
Literals:   0   1   9  11  12  13  14  24
............................................................
Number of Columns=  8
Number of Rows=  9


############################################################
        S W I T C H    T A B L E   R E D U C T I O N
        PASS# 5
############################################################

ESSENTIAL IMPLICANTS DETECTION
NO Essential Implicants Detected.

ROW DOMINATION DETECTION
Imp# 23 is Dominated by Imp# 16 ***>  Implicant # 23 removed.


------------------------------------------------------------
                S W I T C H    T A B L E
                Implicants versus Literals
------------------------------------------------------------
Implicant#  10:    01010000
Implicant#  11:    00110001
Implicant#  12:    00000011
Implicant#  15:    00011100
Implicant#  16:    10010000
Implicant#  19:    01000100
Implicant#  20:    01101000
Implicant#  25:    00110010
```

```
.....................................................
Literals:   0   1   9  11  12  13  14  24
.....................................................
Number of Columns=  8
Number of Rows=  8
-----------------------------------------------------

COLUMN DOMINATION DETECTION
Lit# 11 Dominates Lit#  0 ***********>  Literal # 11 removed.


-----------------------------------------------------
               S W I T C H     T A B L E
               Implicants versus Literals
-----------------------------------------------------
Implicant#  10:    0100000
Implicant#  11:    0010001
Implicant#  12:    0000011
Implicant#  15:    0001100
Implicant#  16:    1000000
Implicant#  19:    0100100
Implicant#  20:    0111000
Implicant#  25:    0010010
.....................................................
Literals:   0   1   9  12  13  14  24
.....................................................
Number of Columns=  7
Number of Rows=  8


#####################################################
        S W I T C H   T A B L E   R E D U C T I O N
        PASS# 6
#####################################################

ESSENTIAL IMPLICANTS DETECTION
Essential Implicants FOUND: [Column#0, Row#4]
Removed Column#  0: Literal#    0
Removed    Row#  4: Implicant#  16


-----------------------------------------------------
               S W I T C H     T A B L E
               Implicants versus Literals
-----------------------------------------------------
Implicant#  10:    100000
Implicant#  11:    010001
Implicant#  12:    000011
Implicant#  15:    001100
Implicant#  19:    100100
Implicant#  20:    111000
```

```
Implicant#  25:    010010
.....................................................
Literals:   1   9  12  13  14  24
.....................................................
Number of Columns=  6
Number of Rows=  7
-----------------------------------------------------

ROW DOMINATION DETECTION
Imp# 10 is Dominated by Imp# 19 ***>  Implicant # 10 removed.


-----------------------------------------------------
               S W I T C H     T A B L E
               Implicants versus Literals
-----------------------------------------------------
Implicant#  11:    010001
Implicant#  12:    000011
Implicant#  15:    001100
Implicant#  19:    100100
Implicant#  20:    111000
Implicant#  25:    010010
.....................................................
Literals:   1   9  12  13  14  24
.....................................................
Number of Columns=  6
Number of Rows=  6
-----------------------------------------------------
COLUMN DOMINATION DETECTION
NO Dominating Columns Detected.


#####################################################
        S W I T C H    T A B L E   R E D U C T I O N
        PASS# 7
#####################################################

ESSENTIAL IMPLICANTS DETECTION
NO Essential Implicants Detected.

ROW DOMINATION DETECTION
NO Dominated Rows Detected.

COLUMN DOMINATION DETECTION
NO Dominating Columns Detected.
```

```
##########################################################
          COMPLETE SWITCH TABLE REDUCTION PROCESS
##########################################################

IMPLICANTS MUST BE INCLUDED IN THE FINAL SOP:
     Implicant#:    2
     Implicant#:   16
     Total Essential Implicants FOUND = 2

----------------------------------------------------------
                S W I T C H     T A B L E
                Implicants versus Literals
----------------------------------------------------------
Implicant#  11:    010001
Implicant#  12:    000011
Implicant#  15:    001100
Implicant#  19:    100100
Implicant#  20:    111000
Implicant#  25:    010010
..........................................................
Literals:    1   9  12  13  14  24
..........................................................
Number of Columns=  6
Number of Rows=  6
----------------------------------------------------------


##########################################################
            BREADTH FIRST SEACH IN OPERATION
##########################################################

A SOLUTION has been FOUND! ===>    PMP Node#  1 is CREATED.

FINAL SOP:
     SOL#  1: Cost =  3
              Implicants:     12   15   20
     Total SOLUTIONS Found = 1

----------------------------------------------------------
IMPLICANTS MUST BE INCLUDED IN THE FINAL SOP:
     Implicant#:    2
     Implicant#:   16
     Total Essential Implicants FOUND = 2

.......................................
:  NOTE:  THE FINAL SOP EXPRESSION    :
:         MUST INCLUDES THE ABOVE  2  :
:         ESSENTIAL IMPLICANTS        :
:.....................................:

TOTAL EXECUTION TIME = 181.000000 [Secs/100]
```

```
##################################################
                  D A T A F I L E
##################################################

 23  16
9999
 22  17   3  20  19  10
9999
  2  18  16  14  15  13   6  11
9999
  2
9999
 25  24   2  17  16   9   4   1
9999
 14  21  17  10   8   3   4  12  11  23
9999
  2   3  18  11  16
9999
 25  20  10  11   2
9999
  9  14   6   2  22  15   7   8
9999
 11  20  22  25
9999
  1  13  20  17  23  15  21  24  16
9999
 25  11   5  16  10  15   6
9999
  6  15   1  17  20
9999
 19   3  15
9999
  8   7  25  12
9999
 23  10  24  13  14  15   3  16   6   2   5  11  22
9999
 17  22  24  13   7  10  18  21   2   9  25   3   4  15
9999
 25  17  18  20   7   9  15   3   5  21  14  16   8
9999
 11  24  22  13   6  12  15  20  16   7  19  10
9999
 23  25  21  20   6   8  10  13   7   4  24  16  17
9999
 23  15  25  21  12  18  10  24   9  14
9999
  1  23   2   4  10  20  12  21  16  25
```

```
9999
   4   7  20   2
9999
   4   9   7  14  21  23   2  24   6  10  12
9999
  14  11  24  12  21 9999
```